Chapter 3

Computing posteriors

One of the primary advantages of doing Bayesian statistics is the flexibility to design the analysis to fit the problem, rather than to coerce the problem into an existing routine in your statistics package. The downside to this is that, once you have designed the model to fit to the data, there may not be a routine you can use directly to fit it. Searching the parameter space to fit a model is more taxing in the Bayesian paradigm, because the *whole* posterior is of interest, while in contrast most classical analyses split the task into two (fairly) simple steps: finding the maximum of the likelihood function (say, using Newton–Raphson) and then plugging the estimate in to the Fisher information to derive an asymptotic confidence interval.

Needless to say, there are several approaches that are widely used to search the parameter space, evaluating the posterior along the way. We will cover four methods—grid search, Monte Carlo, importance sampling, and Markov chain Monte Carlo—of which the latter two are the most useful for realistically difficult applications.

3.1 Grid search/brute force

We have actually already introduced grid searching in chapter 1, with the `R` code that derived the posterior for the Thai AIDS trial example, i.e.

```
pv    = seq(0.00001,0.05,0.00001)
xv    = 51; nv = 8197
logf1 = xv*log(pv) + (nv-xv)*log(1-pv)
f2    = exp(logf1-max(logf1))
intf2 = sum(f2)*(pv[2]-pv[1])
post  = f2/intf2
```

This took a subset, $[0.00001, 0.05]$, of the parameter support, $[0, 1]$, in which we thought the parameter lay, and split it up into bins. The posterior (to a constant of proportionality) was evaluated at each point on the grid and, making the assumption that the posterior does not change much from bin to bin, the integral of the posterior can be approximated by the sum of the area of rectangles with width the inter-bin spacing and height the provisional posterior (to a constant), allowing the approximate posterior to be scaled appropriately to integrate to unity. In performing this calculation the decision has to be made as to how to set up the grid—in an earlier version, I tried

```
pv    = seq(0,1,0.01)
```

but the grid was too coarse and many points were far from the values given high weight by the likelihood, leading to subsequent refinements.

Note that if there are multiple parameters, then the grid must have multiple dimensions, with several deleterious effects. More function calls will be needed, for a start. If 100 grid points are satisfactory to evaluate a single parameter over a plausible support, then $100^2 = 10\,000$ are needed for two parameters, $100^3 = 1\,000\,000$ for three parameters, and many more for even moderately sized problems. Each function call takes time, and although this time is (often or usually) negligible if a thousand or hundred thousand calls are made, it certainly is not for millions, billions or more. Even if the time needed is manageable, values need to be stored to be used for subsequent analysis. If the space needed exceeds your available RAM, then either the computer will crash, or it will start shunting information to the hard disk, which slows things down even more. One option is to reduce the fineness of the grid's spacing, so that only 20 (say) points span the plausible support, perhaps also shrinking in the minima and maxima of the grid's range, and hope that the consequent loss in accuracy caused by the resulting truncation is minimal.

I rarely use grid searches for more than two or three dimensions for these reasons. Let us see an application to a two-dimensional problem.

3.1.1 Example: Influenza A (H1N1) serology in Singapore

As mentioned in the previous chapter, influenza pandemic emerge at a frequency of around twice a century. The most recent influenza pandemic emerged in 2009 in Mexico. To derive estimates of severity, and plan vaccination campaigns, it was of interest to determine the proportion of people infected during the first wave, which in Singapore lasted from June to October 2009. Because many infections are asymptomatic, to determine the

proportion infected, we took serological samples from a pre-existing cohort study in Bukit Batok at the beginning, middle, and end of the outbreak and compared the antibody levels at these time points, using evidence of a four-fold or more rise in antibody titres—termed a *seroconversion*—as evidence of infection. This threshold is commonly selected as it is very specific (we shall assume 100% so) and has high sensitivity (but is not perfect). To analyse these data requires, therefore, accounting for the sensitivity of the test.

Let N be the cohort size and S the number who seroconverted. Let p be the probability of infection (which we wish to determine) and σ the sensitivity, i.e. the probability of a positive test given infection. As we wish to estimate p using these data, let us take a uniform prior on $[0, 1]$ for this parameter. We will consider two possible priors for σ, both independent of p. The first is a $U(0, 1)$ or $\mathrm{Be}(1, 1)$, i.e. non-informative. The second is a $\mathrm{Be}(630, 136)$, which we have derived from a previous study by Zambon et al (Ref. [8]) in which 629 patients out of 764 with virologically confirmed influenza infection seroconverted.

The model is thus:

$$S \sim \mathrm{Bin}(N, p\sigma) \tag{3.1}$$
$$p \sim \mathrm{Be}(1, 1) \tag{3.2}$$
$$\sigma \sim \mathrm{Be}(a, b) \tag{3.3}$$

where (a, b) is either $(1, 1)$ or $(630, 136)$, depending on the prior we are using.

`R` code to evaluate the posterior might look as follows:

```
a=1;b=1 # prior 1
S=98;N=727
pV=seq(0.01,0.99,0.01)
sigmaV=seq(0.01,0.99,0.01)
logpostM=array(0,c(length(pV),length(sigmaV)))
for(i in 1:length(pV))
{
  logpostM[i,]=dbinom(S,N,pV[i]*sigmaV,log=TRUE)+
               dbeta(rep(pV[i],length(sigmaV)),1,1,log=TRUE)+
               dbeta(sigmaV,a,b,log=TRUE)
}
postM=exp(logpostM-max(logpostM))
postM=postM/(sum(postM)*(pV[2]-pV[1])*(sigmaV[2]-sigmaV[1]))
```

The two posteriors are presented in figure 3.1. The non-informative prior for σ yields a useless posterior for p (posterior mean 43%, 95%I 13–95%),

because there isn't enough information content in the data for both parameters to be estimated, only their product. The informative prior yields a fairly precise posterior for p (posterior mean 17%, 95%I 13–20%) and a posterior for σ that, essentially, equals the prior.

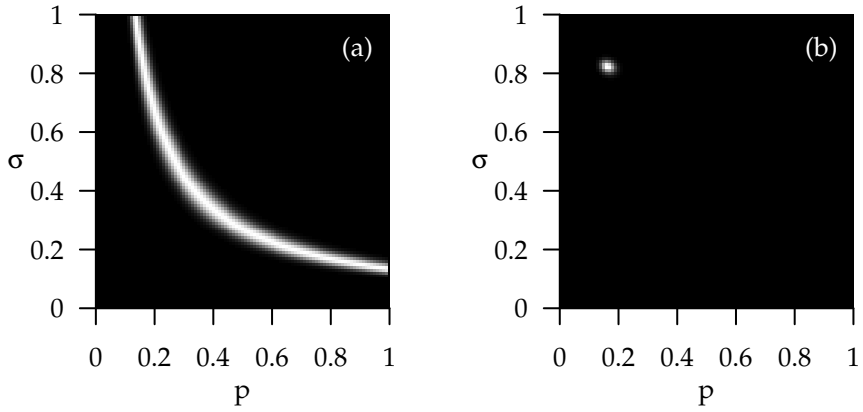

Figure 3.1: **Posterior distribution of probability of infection (p) and sensitivity of four-fold rise in antibody titres in sera (σ) under two prior distributions.** (a) Uniform priors on (0,1) for both p and σ. (b) Beta(630,136) prior for σ, derived from data in ref. [8].

3.2 Monte Carlo sampling

Imagine that you were interested in the size of elephant tusks in the population of wild African elephants (perhaps because you are interested in the effect of poaching on selection pressure for small tusks)—and that you had enough money to fly to Africa. How would you do it? One approach would be to find elephants (preferably with a well designed sampling protocol) and measure their tusks—the larger the sample of tusks you measured, the closer the empirical distribution of tusk lengths to the corresponding unknown population distribution. Time or money limitations would inevitably limit the number of samples you could take—at which point you would use statistical methods to work out plausible values for characteristics of the population distribution, such as its mean—but were money no concern, you could keep sampling until your sample was so large it effectively *was* the population, and you wouldn't need to do any statistical inference anymore, because whatever characteristics of the population you were interested in could be obtained just by inspecting the sample.

In Bayesian statistics, there is likewise an unknown distribution for the parameters θ, and you wish to characterise that distribution. If it were possible to draw a sample **of the parameters from their posterior distribution**, and to make that sample very very big, then the sample you thus obtained would be sufficiently close to the actual distribution that it could be used **as if it were the actual distribution**, for example by calculating the sample mean and equating it to the posterior mean, or finding the 2.5 and 97.5%iles and equating these to the equal tailed 95% uncertainty interval. If your sample were as large as an elephants, you wouldn't need to do any statistical inference with it anymore.

This idea is called Monte Carlo sampling, named after the capital of Monaco, the Mediterranean principality famous for its casinos and glamour. More formally, if θ_i is the ith independent draw from $\mathrm{p}(\tilde{\theta} = \theta_i|\mathrm{data})$ and

$$\hat{g}_m(\theta) = \sum_{i=1}^{m} g(\theta_i)/m \tag{3.4}$$

is the Monte Carlo estimate of the posterior of the function $g(\theta)$, then

$$\hat{g}_m(\theta) \to \int g(\theta)\mathrm{p}(\theta|\mathrm{data})\,\mathrm{d}\theta \tag{3.5}$$

as $m \to \infty$. Note that $g()$ could be any function, such as the identity function (for the mean), or square (to get the variance).

To use Monte Carlo sampling in its simplest form, you need to be able to work out the form for the posterior in a way that permits simulation. This may be possible for some simple examples for which a conjugate prior can be found. Let us see an example in the next section. For more complicated models, in which the posterior's form is non-standard and does not belong to a simple parametric family of distributions, you do not know what distribution to sample from and so must 'guess' the distribution and then 'correct' it to make the sample fair. Two methods to do just this are described later in the chapter.

3.3 Example: Thai HIV vaccine trial

The oft-aforementioned Thai HIV vaccine trial was not performed to estimate the proportion of vaccinees infected after 3.5 years—it was to *compare* the proportion infected on the vaccinated and unvaccinated arms. If we assume independence in infection status between individuals within and between arms, and take a prior distribution $\mathrm{p}(p_v, p_u) = \mathbf{1}\{(p_v, p_u) \in [0,1]^2\}$, i.e. both

proportions are *a priori* uniform on $[0, 1]$ and independent, then the posterior factorises nicely, as follows (note that I'm dropping the dependency on N_u and N_v for brevity):

$$\begin{aligned} \mathrm{p}(p_v, p_u|X_v, X_u) &\propto \mathrm{p}(X_v, X_u|p_v, p_u)\mathrm{p}(p_v, p_u) && (3.6)\\ &= \mathrm{p}(X_v|p_v, p_u)\mathrm{p}(X_u|p_v, p_u)\mathrm{p}(p_v)\mathrm{p}(p_u) && (3.7)\\ &= \mathrm{p}(X_v|p_v)\mathrm{p}(X_u|p_u)\mathrm{p}(p_v)\mathrm{p}(p_u) && (3.8)\\ &\propto \mathrm{p}(p_v|X_v) \times \mathrm{p}(p_u|X_u) && (3.9) \end{aligned}$$

both of which are beta, independently. So the posterior distribution of p_v is $\mathrm{Be}(52, 8197)$ independently of p_u which is, *a posteriori*, $\mathrm{Be}(75, 8199)$.

Now, we could do a grid search here (and this would be useful practice/homework) but we could also just simulate directly from these two beta distributions, as `R` has a built-in routine to sample betas quickly. `R` code to do this follows.

```
xv=51;nv=8197
xu=74;nu=8198
M=10000
pv=rbeta(M,1+xv,1+nv-xv)
pu=rbeta(M,1+xu,1+nu-xu)
summariser=function(x)
{
   cat('Posterior mean:',signif(mean(x),3),
       '95%I:',signif(quantile(x,c(0.025,0.975))),
       '\n')
}
summariser(pv)
summariser(pu)
summariser(pv/pu)
summariser(pu-pv)
mean(pu<pv)
```

The summariser function prints the mean and 95% equal tailed interval for its vector argument. Note that the posterior distribution of functions of the parameters can readily be obtained once you have samples from the posterior, but that you must retain the indexing (so don't scramble the order of one or more of the parameter sample vectors). This is a substantial benefit of a Bayesian approach, because in the classical paradigm, obtaining confidence intervals of functions of parameters, such as the ratio of means from two normal distributions, requires tedius calculus and approximations based on the delta method.

In the last line of code, we calculate the proportion of samples for which one parameter is bigger than the other. In notation, this is

$$\mathrm{p}(\tilde{p}_u < \tilde{p}_v | X_u, X_v, N_u, N_v) \tag{3.10}$$

i.e. the posterior probability that the proportion infected in the unvaccinated group is less than that of the vaccinated group. Note the distinction between this—the probability an hypothesis is true given the observed data—with the definition of a *p*-value—the probability of imaginary data that were not observed but would have been more extreme than the actual data given that the hypothesis were true. The classical approach to hypothesis testing involves taking, typically, a point null hypothesis, that in most cases would not plausibly be true anyway (for instance, in an observational study, assuming a covariate has no association with the outcome after adjusting for a small number of other covariates), assuming it to be true, and calculating the probability of non-observed data, and using that to try to demonstrate that the null hypothesis is wrong. The Bayesian approach can naturally assess the evidence for any non-point hypothesis by simply calculating the probability the hypothesis is true. For point hypotheses, if the prior assigns zero probability to their being true, are impossible *a posteriori*. Given that there are very few instances in which a nil hypothesis is likely to be true anyway (two examples might be in a randomised clinical trial of one placebo against another, or in evaluating claims of scientific fraud in which patients were not randomly allocated to treatment arms), I find this to be not a great weakness, though others might.

3.4 Importance sampling

The problem with Monte Carlo sampling in its most basic form is that to use it you need to (i) know the specific distribution of the posterior of the parameters and (ii) be able to simulate this distribution. More usually, however, you know the distribution only in the sense that you can calculate it for any given values of the parameters (perhaps up to a constant of proportionality) but cannot simulate from it, so rather than knowing that $p_v|X_v, N_v \sim \mathrm{Be}(52, 8147)$, say, all you know is that $\mathrm{p}(p_v|X_v, N_v) \propto p_v^{51}(1-p_v)^{8146}$. This lack of knowledge means you do not know how to select values for the parameters, as you could for basic Monte Carlo sampling.

The idea behind importance sampling is that instead of sampling from the posterior itself, whose form you don't know, you simulate from a different distribution—which I will call the proposal distribution to make an analogy clearer later—that, you hope, is close to the actual posterior, and "correct"

it to account for the fact that it is not the actual posterior. The correction is done by weighting the sampled points by the ratio of the posterior to the proposal density, both of which you should be able to evaluate numerically (if you cannot evaluate the proposal density, choose one you can).

The algorithm is as follows.

1. For $i \in \{1, 2, \ldots, m\}$:
   (a) Draw θ_i from the proposal distribution with density $\pi(\tilde{\theta} = \theta_i)$;
   (b) Calculate the log prior plus log likelihood at this point, i.e. $\log \mathrm{p}(\theta_i) + \log \mathrm{p}(\mathrm{data}|\theta_i)$;
   (c) Calculate the log importance weight

$$\log w_i = \log \mathrm{p}(\theta_i) + \log \mathrm{p}(\mathrm{data}|\theta_i) - \log \pi(\theta_i); \tag{3.11}$$

2. Scale the weights so they sum to unity.

If the proposal distribution is exactly the posterior distribution (by chance or design) then the weights will all be equal and the routine reduces to a Monte Carlo sample. If the proposal density is lower in one part of the parameter space than the posterior, then the routine samples too few points from that region, and these points have to be weighted more highly, and vice versa, if the proposal oversamples one particular part of the parameter space, then those sampled points will have lower weight to compensate.

This routine will be optimal if the proposal distribution is equal to the posterior, and as the proposal becomes worse and worse, the distribution of weights becomes more and more heterogeneous, with the result that the sample is less and less efficient (to see why, imagine a sample with one point with weight one and 999 points with weight 0: is this more like a sample of size $n = 1$ or $n = 1000$?). This can be characterised by the *effective sample size* of the weighted sample. The effective sample size is defined [14] to be

$$ESS = \frac{\left(\sum_{i=1}^{n} w_i\right)^2}{\sum_{i=1}^{n} w_i^2}. \tag{3.12}$$

If all n draws have equal weight then the ESS is n; otherwise, $ESS < n$.

How does one choose the proposal distribution to use for any particular application? One might think that the prior is a natural choice. In a way, it is, for it has the same support as the posterior (assuming you selected it to), and the calculations simplify as the priors cancel on top and bottom of the weight ratio, leaving just the likelihoods. However, if the data are very informative, then the likelihood will be peaky (or, the posterior and the

prior will be very different), and so most proposed values will receive very low weights and the sample will be inefficient.

Alternatives are to approximate the posterior by a multivariate normal (or, even better, t) with mean the maximum likelihood estimate and covariance the covariance matrix, if the problem has a simple expression for these. This might be a valuable strategy if the sample size is too small to justify using asymptotic results to obtain classical confidence intervals. You might also just use trial and error, rerunning the routine with better choices if the initial proposal is not effective.

Finally, the idea of trial and error could be formalised by an iterative procedure, in which a *scaled* log-posterior is used for initial runs (perhaps by using a random subset of the data, or using all the data but multiplying the log-likelihood or log-posterior by a factor, the latter equivalent to raising the likelihood to a power < 1), a multivariate normal distribution is fitted to the resulting weighted sample, and the multivariate normal is used as the proposal for the next iteration. On the final iteration, the log-posterior is no longer scaled, yielding a correctly weighted sample.

Let us apply importance sampling to two examples we have used grid searches on.

3.4.1 Example: HIV trial in Thailand

We can treat each arm separately for the reasons described before. Let us focus on the vaccine arm. The prior is $\text{Be}(1, 1)$ and the support of the parameter p_v is the interval $[0, 1]$. A potential proposal distribution would be $U(0, 1)$, i.e. the prior, because it shares the support and is "neutral." Note that in the description I have called the prior $\text{Be}(1, 1)$ and the proposal $U(0, 1)$ to help distinguish them in the code. An `R` routine follows:

```
xv    = 51; nv = 8197
M     = 10000
pv    = runif(M)
logp = dbinom(xv,nv,pv,log=TRUE)+dbeta(pv,1,1,log=TRUE)
logw = logp - dunif(pv,log=TRUE)
w     = exp(logw-max(logw)); w=w/sum(w)
```

Point and interval estimates can be derived using the following code:

```
mean_pv = weighted.mean(pv,w)
mean_pv2 = weighted.mean(pv*pv,w)
var_pv   = mean_pv2 - mean_pv*mean_pv
sw=w[order(pv)]
```

```
spv=pv[order(pv)]
sCDF_pv=cumsum(sw)
CIL=spv[which.min((sCDF_pv-0.025)^2)]
CIU=spv[which.min((sCDF_pv-0.975)^2)]
```

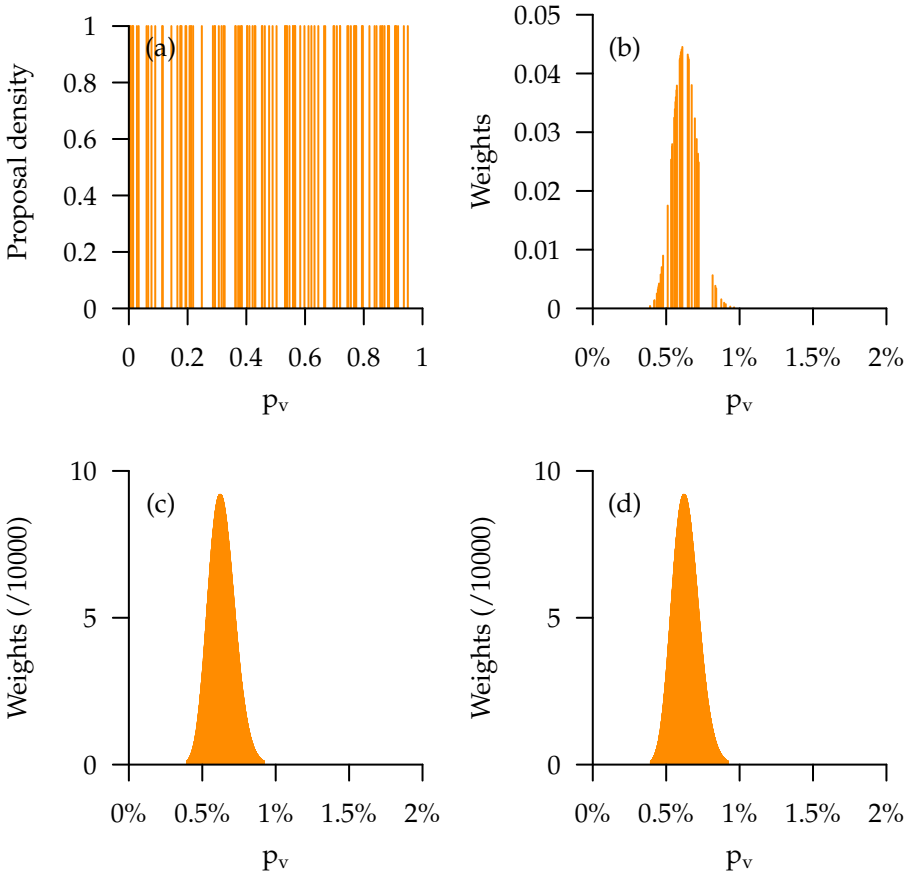

Figure 3.2: **Importance sampling schematic.** (a) shows a (thinned) sample from a $U(0,1)$ distribution used as the proposal distribution. (b) shows the resulting weighted sample. Most points have near zero weight and hence are not visible. (c) and (d) show results using $U(0,0.02)$ and $N(0.006, 0.0007^2)$ proposal distribution, respectively.

The process is plotted in figure 3.2. Only 69 points in this sample have a weight of more than 0.000001, and the effective sample size is 28. This is not really enough to get a good estimate, which is why the point (0.61%) and interval estimate (0.45–0.72%) from the importance sample are somewhat different from those obtained using other methods (0.63%, 0.47–0.82%).

To remedy this, we might try:

- a proposal distribution that is uniform on $[0, 0.02]$, noting some truncation error. This should increase the effective sample size 50-fold (the truncation error can be resolved by proposing from a mix of $U(0, 0.02)$ and $U(0.02, 1)$;
- a normal proposal distribution with mean the point estimate from above and variance derived similarly from the weighted sample;
- a normal proposal distribution with mean the sample proportion and standard deviation equal to the standard error of the sample proportion.

Let us now try these:

```
## Narrower uniform:
pvA   = runif(M,0,0.02)
logpA = dbinom(xv,nv,pvA,log=TRUE)+dbeta(pvA,1,1,log=TRUE)
logwA = logp - dunif(pvA,0,0.02,log=TRUE)
wA    = exp(logwA-max(logwA)); wA=wA/sum(wA)
## ----------------
## Normal using mean, variance estimated from first try
pvB   = rnorm(M,mean_pv,sqrt(var_pv))
logpB = dbinom(xv,nv,pvB,log=TRUE)+dbeta(pvB,1,1,log=TRUE)
logwB = logpB - dnorm(pvB,mean_pv,sqrt(var_pv),log=TRUE)
wB    = exp(logwB-max(logwB)); wB=wB/sum(wB)
## ----------------
## Normal using mean, variance from classical estimates
pvC   = rnorm(M,xv/nv,sqrt((xv/nv)*(1-xv/nv)/nv))
logpC = dbinom(xv,nv,pvC,log=TRUE)+dbeta(pvC,1,1,log=TRUE)
logwC = logpC - dnorm(pvC,xv/nv,sqrt((xv/nv)*(1-xv/nv)/nv),log=TRUE)
wC    = exp(logwC-max(logwC)); wC=wC/sum(wC)
```

Output are presented in table 3.1.

3.4.2 Example: Influenza A (H1N1) serology in Singapore

Let us return to the two-parameter binomial distribution used to model seroconversion to pandemic influenza, i.e. where $S \sim \text{Bin}(N, p\sigma)$. We saw that a non-informative, uniform prior on both parameters led to a useless, banana shaped posterior. For this example we shall use the informative $\text{Be}(630, 136)$

Table 3.1: **Effect of different proposal distributions on estimates, Thai vaccine example.** Effective sample sizes (ESS), posterior means and 95% equal tailed intervals are tabulated.

Method	ESS	Mean	Lower	Upper
Grid	NA	0.63%	0.47%	0.82%
IS, $U(0,1)$	28	0.61%	0.45%	0.72%
IS, $U(0,0.02)$	1542	0.63%	0.47%	0.82%
IS, $N(0.006, 0.0007^2)$	2308	0.63%	0.48%	0.85%
IS, $N(0.006, 0.0009^2)$	9504	0.63%	0.48%	0.85%

prior for σ, though the uniform prior is useful practice for homework. Let us use a uniform proposal distribution for both parameters on the range $[0,1]^2$ as a first attempt.

```
S     = 98
N     = 727
M     = 10000
p     = runif(M)
sigma = runif(M)
logp  = dbinom(S,N,p*sigma,log=TRUE)+
        dbeta(p,1,1,log=TRUE)+
        dbeta(sigma,630,136,log=TRUE)
logw  = logp - dunif(p,log=TRUE) - dunif(sigma,log=TRUE)
w     = exp(logw-max(logw)); w=w/sum(w)
```

This gives an ESS of merely 22, indicating very poor estimates. We can improve this by taking a smarter proposal distribution for σ, since we have an informative prior distribution and expect the posterior may resemble this. The code changes as follows:

```
sigma = rbeta(M,630,136)
logw  = logp - dunif(p,log=TRUE) - dbeta(sigma,100,20,log=TRUE)
```

This improves the ESS (to 458) but not by much, as many proposals to p are made that are wasted. One could manually tweak the proposal for p to get less wastage, or increase the sample size M (since the code is very fast to run), and both would work quite well here. An alternative is to use an iterative approach in which we start with a poor proposal distribution and have the computer refine it sequentially. One set of code to do this follows:

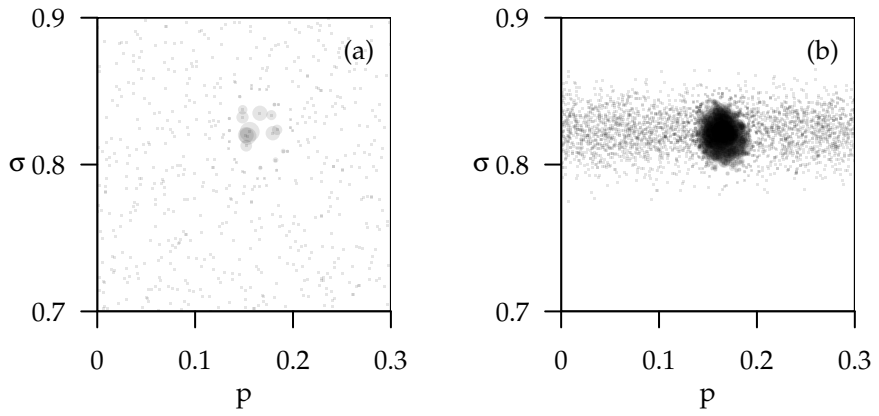

Figure 3.3: **Importance sampling for influenza serology example.** (a) uniform proposal on $[0, 1]$ for both p and σ; (b) beta prior used as proposal for σ. Area of points is proportional to weight.

```
NITERATIONS=10
S = sample(c(rep(1,98),rep(0,727-98)))
N = rep(1,727)
M = 10000
prange     = c(0,1)
sigmarange = c(0,1)
for(iteration in 1:NITERATIONS)
{
  print(iteration)
  p     = runif(M,prange[1],prange[2])
  sigma = runif(M,sigmarange[1],sigmarange[2])
  use   = 1:round(length(N)*iteration/NITERATIONS)
  logp  = dbinom(sum(S[use]),sum(N[use]),p*sigma,log=TRUE)+
          dbeta(p,1,1,log=TRUE)+dbeta(sigma,630,136,log=TRUE)
  logw  = logp - dunif(p,prange[1],prange[2],log=TRUE)-
          dunif(sigma,sigmarange[1],sigmarange[2],log=TRUE)
  w     = exp(logw-max(logw)); w=w/sum(w)
  i          = which(w>(mean(w)*0.05))
  prange     = range(p[i])
  sigmarange = range(sigma[i])
}
```

Note several points about this routine:

- The code proposes values uniformly from a box with end points de-

termined by the previous iteration. The end points are (arbitrarily) selected to be the range spanning all values of p and σ that have weight at least 5% of the mean weight. To select 5%, I looked at the output of the final iteration to confirm that a sufficiently large border surrounded the posterior.

- The likelihood uses a subset of the data on non-final iterations. This makes the likelihood flatter which helps the routine gradually home in on the posterior. It also means that only results from the final iteration can be reported.

The effective sample size for the final iteration is 3194, which is far better than the non-iterative routines with poorly chosen proposals. Plots of this are provided in figure 3.4.

3.4.3 Resampling

If it is desired to have a non-weighted sample, this can be achieved by resampling proportional to the weights. This is trivially done in one dimension:

```
M2 = 10000
pv_resampled = sample(x=pv,size=M2,replace=TRUE,prob=w)
```

and only slightly harder in multiple dimensions:

```
indices = sample(x=1:M,size=M2,replace=TRUE,prob=w)
p_resampled = p[indices]
sigma_resampled = sigma[indices]
```

Such a resampled importance sample can be used as if it were a regular sample with equal weights on each point, although note that the effective sample size does not increase by doing this.

3.4.4 When to use importance sampling

Importance sampling works best when you are able to generate a sample from a distribution that is close to the actual posterior. It is poor otherwise. Unfortunately, in most applied problems, it is hard to identify a sufficiently good proposal distribution. Iteratively generating a series of importance samples, each time closer to the desired posterior, is one way importance sampling can be useful, although this can be a computationally expensive strategy, for it would usually involve discarding intermediate iterations.

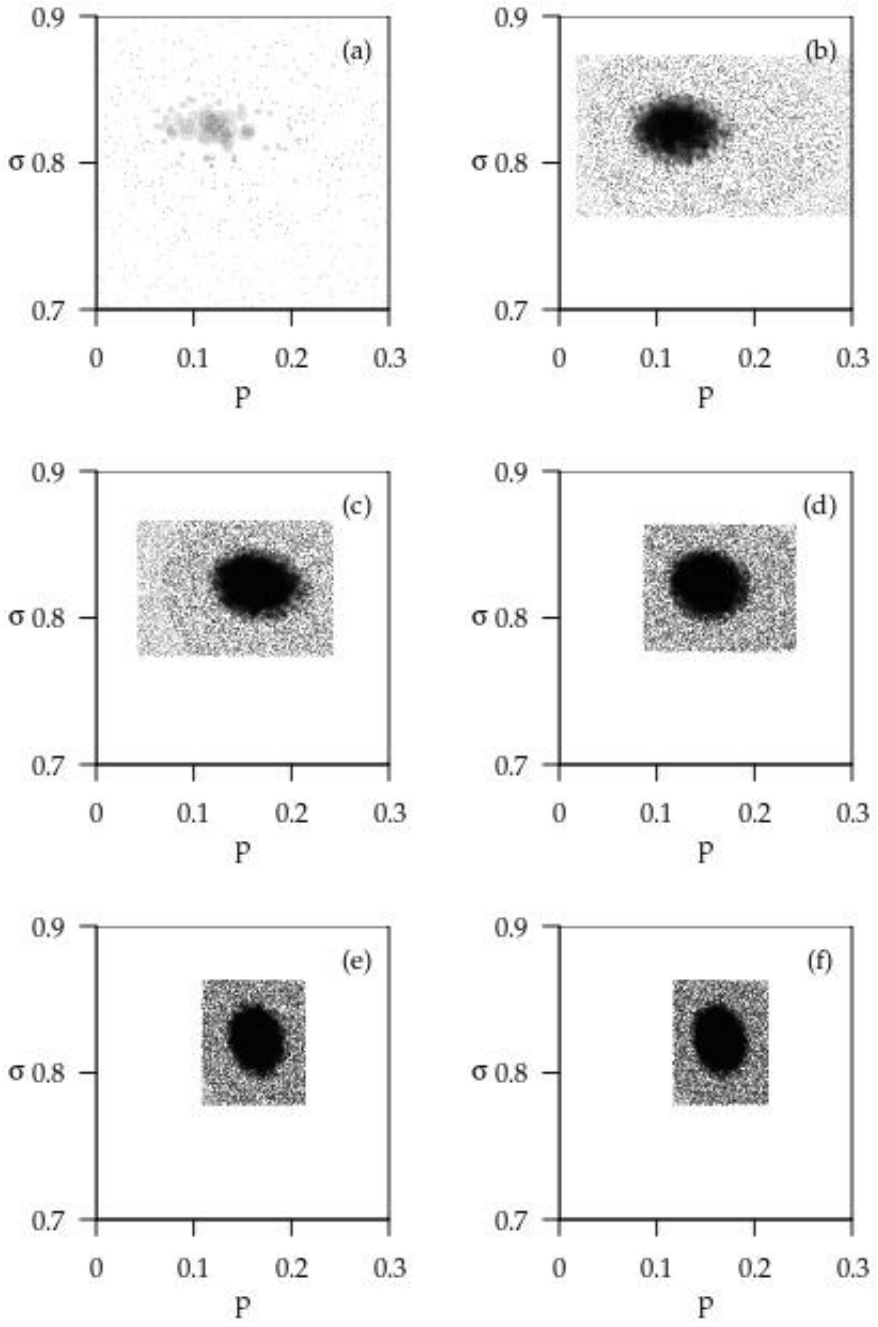

Figure 3.4: **Importance sampling for influenza serology example: iterative proposals.** Iteration 1 (a), 2 (b), 3 (c), 4 (d), 9 (e), 10 (f). Iterations before 10 use subsets of the data, so that the likelihood surface is flatter. Area of points is proportional to weight.

Importance sampling can be very useful when a sample from one posterior is generated and you want to switch priors to get a different posterior (perhaps as a sensitivity analysis to the choice of prior) but do not wish to resimulate the posterior—in this scenario you could propose from the first posterior (from which you already have a sample) and apply a weight based on the ratio of *priors* (rather than likelihoods, as is usual), a calculation that should be rapid.

A variant of importance sampling—sequential importance sampling—can be useful when data arrive over time (say, daily reports from sentinels of disease burden). This is a convenient way to modify (ever so slightly) the current posterior when a new data point arrives. For further details, see ref. [13].

3.5 Markov chain Monte Carlo

Markov chain Monte Carlo, or MCMC, has revolutionised Bayesian data analysis over the last two decades, though its origins lie much earlier, with a paper in the physics literature in the 1950s [10] and another in the 1970s introducing it to the statistics community [11]. MCMC is an extension of Monte Carlo that, like importance sampling, does not require being able to simulate directly from the posterior density, but will "correct" samples from some other distribution. In importance sampling, draws are taken independently of each other from the wrong distribution and then corrected by weighting them to obtain a weighted sample from the correct posterior distribution. In MCMC, draws are obtained by setting up a clever Markov chain that proposes changes to the parameters from one iteration to the next from the wrong distribution and corrects them by counting draws two or more times, by sometimes staying in the same position from one draw to the next, so that the final sample contains equally weighted draws from the posterior.

The ideas in this section are probably the most important in this course and so you are advised to make extra care to understand them.

3.5.1 Refresher on Markov chains

You should already know the basics of Markov chains, and this section is meant just as a refresher. If you do not, please read up on wikipedia or in the library without delay.

A Markov chain is a discrete time stochastic process in which the distribution of the state at time $t+1$, θ_{t+1}, depends on previous states $\theta_t, \theta_{t-1}, \ldots$

only through θ_t, i.e. once you know the present, history becomes irrelevant in predicting the future. Define $\phi(\theta_{t+1}|\theta_t)$ to be the transition probability from θ_t to θ_{t+1}; it is in its most basic form the same for all values of t. Note that the range for θ_{t+1} in this course will usually be some subset of real space, in contrast to the Markov chains defined on discrete sets in most introductory stochastic process courses.

Under some conditions[1] the Markov chain will have a stationary distribution to which it will tend to as $t \to \infty$. If $\pi(\theta)$ is the stationary distribution, then if θ_t comes from $\pi(\theta)$ then so does θ_{t+1}. A sketch of a proof, along with references to full proofs, is provided in ref. [1].

Imagine that you were really, really interested in sampling from the stationary distribution of some particular Markov chain. One way to do it would be to select a value that was roughly near the stationary distribution, then run the Markov chain for enough time (perhaps 1000 iterations?) that it plausibly was now in the stationary distribution, and then take that 1000th value as the first sample from the stationary distribution, and keep on repeating the whole process, taking an initial value, iterating it forward, and storing the 1000th value, until you had a big enough sample for analysis.

Well, if the 1000th sample were from the stationary distribution, then the 1001st sample would be too, as would the 1002nd, and so on, so you wouldn't need to restart the Markov chain every time—instead you could run it to the 1000th iteration, store it, run it one further iteration, store *that*, ..., and keep going one step forward at a time until the desired sample size were reached. This would not give you an independent sample from the stationary distribution, but a correlated one, because the tth and $t+1$th samples would be correlated with each other, but each draw would, indeed, be from the stationary distribution, assuming the first were.

Now imagine that you could design a Markov chain that, instead of just having any old distribution as its stationary distribution, had a stationary distribution that was one and the same as the posterior distribution that you were trying to sample. Then you could simulate a long chain from this process and store the values sampled and then **use that correlated sample as if it were the posterior distribution**. In other words, once the model for data and parameters is decided, the problem of analysis boils down to working out how to set up such a Markov chain. It turns out that setting up a Markov chain with a desired stationary distribution is rather easy to do.

[1] Ergodicity, irreducibility, and aperiodicity—which I am sure you will all remember fondly from previous classes on stochastic processes, but see chapter 2 of ref. [12] if you seek a reminder.

3.5.2 The Metropolis–Hastings algorithm

The Metropolis–Hastings algorithm is the most common MCMC routine in use. It works by proposing a new value of one, several, or all parameters, comparing the difference in log-posteriors between the proposed and the current configurations, and, if the proposal is better, accepting it, while if the proposal has a lower log-posterior, possibly accepting it. The next value of the Markov chain is either the proposed or the current value; because both the proposed value and the accepted value may depend on the current value, subsequent draws are correlated.

The algorithm is pseudo-code is simple. Let θ_i be the value of the parameter (vector) at iteration i and $\theta^\star$ be the proposed value, with $\mathrm{p}(\theta|\text{data})$ the posterior density (up to a constant of proportionality) and $q(\theta^\star|\theta_i)$ the proposal density.

1. Set $i = 0$.
2. Select θ_i such that $\mathrm{p}(\theta_i|\text{data}) > 0$, i.e. such that it is a valid parameter configuration. Calculate the log-posterior.
3. Generate $\theta^\star$ from $q(\theta^\star|\theta_i)$. Calculate $\log \mathrm{p}(\theta^\star|\text{data})$.
4. Let $\log \alpha = \log \mathrm{p}(\theta^\star|\text{data}) - \log \mathrm{p}(\theta_i|\text{data}) + \log q(\theta_i|\theta^\star) - \log q(\theta^\star|\theta_i)$.
5. Generate $u \sim U(0, 1)$. If $\log u \leq \log \alpha$, let $\theta_{i+1} = \theta^\star$; otherwise let $\theta_{i+1} = \theta_i$.
6. Increment i and repeat steps 3–6 until a sample of the desired size is reached.

If the proposal density happened—magically—to be exactly the same as the posterior density, then $\log \alpha = 0$ and $u \leq \alpha$ for all u generated, so that every proposal is accepted and the routine is equivalent to a basic Monte Carlo sample. If not, the Metropolis–Hastings algorithm "corrects" the proposal. A proof of why the Metropolis–Hastings algorithm yields a Markov chain with the posterior is rather involved and not very interesting, so it is skipped.

One of the most amazing things about the algorithm is that it works for any sensible choice of the proposal distribution q. A typical choice is $\theta^\star \sim N(\theta_i, \sigma_\theta)$, i.e. normally distributed (possibly multivariately) around the current value with a standard deviation (or covariance matrix). For a symmetric distribution (around the current value) such as this, $q(\theta_i|\theta^\star) = q(\theta^\star|\theta_i)$ and so these terms disappear from the equation for α, leaving $\log \alpha = \log \mathrm{p}(\theta^\star|\text{data}) - \log \mathrm{p}(\theta_i|\text{data})$, i.e. the log of the ratio of the proposed to current posterior.

3.5.3 Implementation: initial conditions

The initial conditions of the chain are in a sense arbitrary: as long as they are not "illegal" (i.e. with zero probability of generating the observed data), the Markov chain will recover from unrepresentative choices and converge to the posterior, eventually. Starting the chain near the posterior is better for the efficiency of the chain, as it can then spend more time exploring the posterior than trying to find the posterior, but starting the chain far from the posterior makes you more confident that the chain has converged to the posterior and not just a local mode.

In practice, MCMC is often run in a sequence of attempts, with the first run a complete shot in the dark, and the concomitant risk that the initial conditions and proposal distribution are not very efficient. In such cases, the final values of preliminary chains can be useful choices of initial conditions for later runs.

3.5.4 Implementation: proposal distribution

The MCMC sampler will work regardless of the precise choice of proposal distribution. However, some choices of proposal distribution work better than others. There are two main choices:

- A proposal distribution that is centred on the current values, for example, normal. In this case, the decision is (i) whether to update one parameter at a time or all together, and (ii) the (co)variance of the normal distribution. In initial runs it can be a good idea to update one parameter at a time, guessing a standard deviation you think will be suitable. If the standard deviation is too small, then proposals will be to values close to the current parameters and so the Markov chain will explore the posterior distribution very slowly. This is inefficient as it means a great many draws need to be made to get a reasonable effective sample size. If, on the other hand, it is too large, then most proposals will be to values far far away from the posterior distribution, and hence will be rejected, leaving the Markov chain sitting in the same location for many iterations at a time. This, too, is inefficient. Instead, a Goldilocks spot leads to optimal movement around the posterior.
  Proposing a change to all parameters at one go means you need to have a good idea of the appropriate size of the jumps in each direction, because if you make overly large proposed jumps to just one of them, the entire chain will not mix well. Hence it is better to save multivariate proposals to a second or third attempt once the approximate size and

shape of the posterior distribution are established. An effective strategy is to calculate the sample covariance matrix from a dummy run and scale that by a factor (of perhaps 0.5) and use that as the covariance in a multivariate normal proposal.

- The alternative is to use a proposal distribution that is independent of the current values of the chain, perhaps one that has been approximated already using classical methods or a previous run. If in a previous run you estimated the mean and covariance and determined that the posterior was approximately multivariate normal, then you could propose values from a multivariate normal or t distribution with that mean and covariance. Note that this can be a little risky if the covariance or the tails are underestimated, as when proposals are made out into the tails, it can be hard to move away again.

3.5.5 Implementation: burn-in and chain length

Two related issues are how long to run the MCMC sampler, and how long until the MCMC sampler converges away from its initial value and into the posterior distribution. To account for the risk that the initial draws are not representative of the posterior, it is usual to discard the first part of the MCMC sample—this is called the burn-in. Plotting parameter values against interation (the *trace plot*) will give a visual indication of how many samples need to be discarded. Opinion on a rough ball park is divided: Gelman et al [1] recommend discarding a full half, while Gilks et al [12] a mere 1–2%. My experience aligns with Gilks et al: usually I find the MCMC has converged to its ostensibly stationary distribution rather quickly, and if not, the sampler needs to be tweaked or run longer.

How long to run the sampler to get usable estimates? There are two points you should bear in mind. One is that an MCMC sample is inherently less efficient than an independent sample, because the correlation between successive values reduces the effective sample size. Therefore you may need to run it longer than your intuition suggests. The other is that running an MCMC sampler for a long time is much, much less costly than actual data collection, so I find it somewhat disrespectful to run a short MCMC routine when someone has spent days or weeks collecting data, especially if the data come from patients who volunteered to participate in a clinical trial. I usually target 10 000 draws unless the problem is horrendously computationally expensive.

3.5.6 Implementation: number of chains

It can be useful to run several chains rather than a single one. There are three reasons for so doing.

- Running multiple chains allows the chains to be compared. If they start off in different parts of the parameter space and converge to the same place, it provides more confidence that that is, really, the posterior distribution. This also indicates a suitable burn-in length.

- Related to this, if there are multiple posterior modes, it is likely a single chain will not find them all and will instead settle in one. Having multiple chains increases the chance that more than one will be visited, alerting you to the problem (though not really solving it).

- Most computers (at the time of writing) have multiple processors on the motherboard. This means that one computer can finish one MCMC chain of length 10 000 in the same time as it can finish three, say, of the same length. Running several chains in parallel therefore allows an effective doubling or more of computing speed. If you choose to do this, you (i) need to be aware how many processors you have and budget accordingly (for instance, if you run windows it will gobble up an entire processor to keep going [it seems] while you may need a second for other processes such as the internet and word processing, and so should use two fewer chains than processors; if you use linux then you can run the samplers in `nice` mode and run on all processors).

Note that if you do start several chains, do not whatever you do run multiple chains with the same starting conditions and same random number seed unless you want to feel stupid later.

3.5.7 Implementation: assessing convergence

Although in theory (under some usually met conditions), the Markov chain created in an MCMC routine will converge to the correct (posterior) distribution as the number of iterations tends to infinity, you can never be sure that the number of iterations in your routine is enough to create a sample from the posterior and not merely an approximation to the posterior. To reassure yourself, and anyone with a stake in your analysis, that the sample from the MCMC routine really does come from the desired posterior, the convergence of the chain should be assessed. There are various ways to do this.

- **Trace plots:** after the routine has finished, plot individual parameters against iteration number. If the resulting trace plot shows a sample that is slowly wobbling around or moving erratically, then convergence is in doubt. See figure 3.5. If the trace plot shows the parameter making lots of small steps, then you may wish to increase the proposal variance for that parameter and rerun; if it rarely moves but when it does move it moves a long way, then you might rerun with a smaller proposal variance. If two or more parameters appear to move in tandem, then you might inspect their joint distribution (by plotting one against the other). If this indicates the twain are correlated, you may have to propose changes to them jointly, rather than individually. Rerunning with a better proposal distribution may fix any of these problems and yield a sample that appear to have converged.

- **Run several chains:** if done from different positions, and all converge to the same distribution, then this increases confidence that that distribution is actually the posterior.

- **Test stationarity:** One of the earliest convergence diagnostics proposed is due to Geweke [9]. Geweke's diagnostic considers one parameter at a time, taking the first part (post burn-in) and last part of the sample, calculating their means and spectral densities, and calculating a Z score, which should be standard normal if the distribution in the first part of the chain is the same as in the last part (which, unlike most hypotheses tested statistically, is actually plausible). The diagnostic is implemented in the `coda` package in `R`. To use Geweke's diagnostic, one must specify the proportion from each end used in the test: the `coda` default is the first 10% and last 50%. Example syntax, assuming `theta` is a vector sampled using an MCMC sampler, is `library(coda);geweke.diag(theta)`. The function returns the Z scores and fractions used.

3.5.8 Implementation: storage

In running an MCMC sampler, you need to store the values the chain visits somewhere to use them for subsequent analysis. You will sometimes read in the literature suggestions to *thin* your chains by keeping every 10th iteration, say, and discarding the rest. Do this only if you have memory constraints, for instance if you have thousands of parameters to store and your RAM is going to run out. Otherwise it is misguided. Some researchers think successive values need to be independent to be valid (not true) and that a

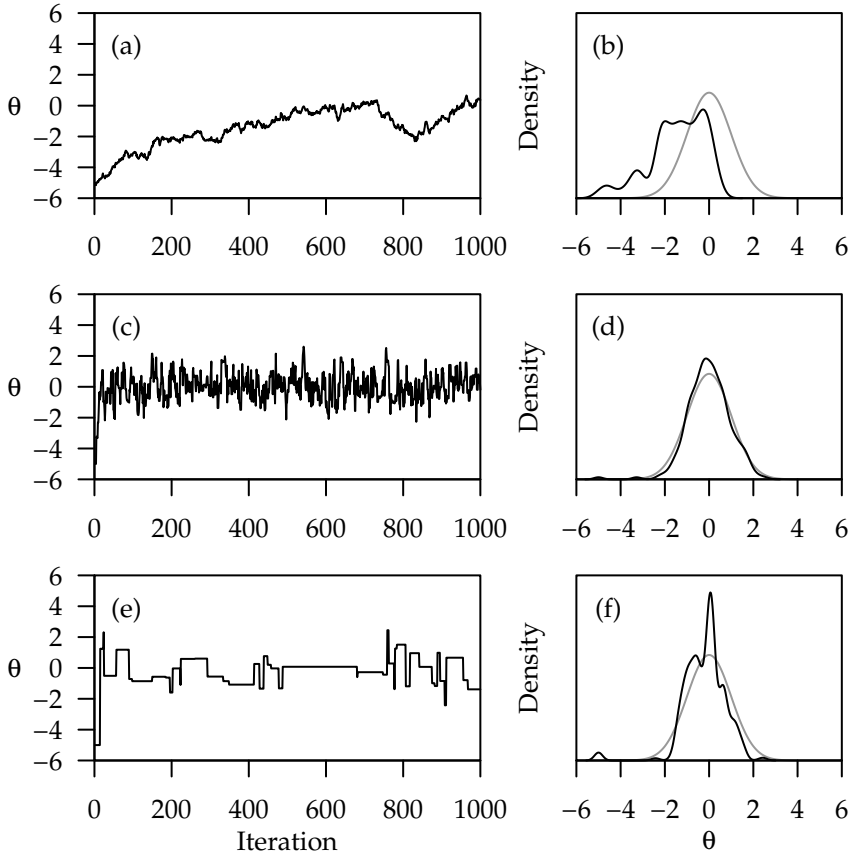

Figure 3.5: **Trace plots to assess convergence.** In all cases, the target distribution is $\mathrm{N}(0, 1)$ (in grey on panels b, d and f). Panels (a) and (b) show the chain that results from a proposal distribution that is too narrow ($\theta^\star \sim \mathrm{N}(\theta_i, 0.1^2)$): most proposals are accepted but the chain does not move quickly around the standard normal (a shows the traceplot, b an estimate of the density after 1000 iterations). Panels (c) and (d) show a chain with a well chosen proposal distribution, normal centred on the current value with standard deviation 1. Panels (e) and (f) show a chain resulting from an overly broad proposal distribution (standard deviation 30). The chain quickly moves to the region of the parameter space supported by the posterior but most subsequent proposals are rejected, being too far from the posterior. In all cases, the starting value was taken to be -5.

chain of 100 000 keeping every 100th draw provides better estimates than a chain of 100 000 keeping every draw (ditto).

Store parameter values of anything you might wish to analyse and the log-posterior (perhaps split into log-likelihood and log-prior).

3.5.9 Example: HIV trial in Thailand

Let us return to our simplest example, the HIV vaccine trial in Thailand, once again. We outline some `R` code that will implement an MCMC sampler which uses a normal distribution centred on the current value of p_v as the proposal distribution. There are several choices to make:

- The initial value of p_v. Since it is a probability, we can choose any value on $[0, 1]$. Let us use 0.5 for illustration. (A more sensible value would be 0.01, as around 1% were infected empirically.)
- The proposal standard deviation. Let us use 0.01, again for illustration.
- The burn-in length and number of MCMC draws. Normally I might use a burn-in of 1 000 and 10 000 draws on a first attempt, but because I wish to demonstrate the behaviour of the chain even at early iterations I will use a burn-in of length 0, i.e. no burn-in. This is usually not recommended.

The code will use the following function, defined first.

```
logposterior=function(pv,xv,nv)
{
  dbinom(xv,nv,pv,log=TRUE)+dunif(pv,log=TRUE)
}
```

The main routine is as follows.

```
set.seed(666)
pv=0.5
xv=51;nv=8197
logpost=logposterior(pv,xv,nv)
MCMCiterations=10000
pvstore=logpoststore=rep(0,MCMCiterations)
for(iteration in 1:MCMCiterations)
{
  if(iteration%%100==0)print(iteration)
  pvold=pv
```

```
  logpostold=logpost
  pv=rnorm(1,pv,0.01)
  REJECT=FALSE
  if(pv<0)REJECT=TRUE
  if(pv>1)REJECT=TRUE
  if(!REJECT)
  {
    logpost=logposterior(pv,xv,nv)
    logaccprob=logpost-logpostold
    lu=log(runif(1))
    if(lu>logaccprob)REJECT=TRUE
  }
  if(REJECT){pv=pvold;logpost=logpostold}
  pvstore[iteration]=pv
  logpoststore[iteration]=logpost
}
```

Note that I have set the seed to a specific number to allow reproduction. This is good practice when you may need to reproduce something precisely later, for instance, in debugging or sharing code with others.

Output is presented in figure 3.6. The sampler takes around 150 iterations to move down to what appears to be the stationary distribution, so a burn-in of length 200 would be appropriate. Thereafter it cycles through the posterior quite well, so the proposal standard deviation appears appropriate. Geweke's diagnostic, after discarding the first 200 iterations, gives a Z score of 0.33, which is consistent with a standard normal. We would therefore be satisfied with the estimates from this sampler, which can be obtained using the same `summariser` function used earlier for Monte Carlo samples.

3.5.10 Example: influenza A (H1N1) serology in Singapore

We will set up an MCMC sampler to explore the distribution of p and σ with a non-informative uniform prior on $[0, 1]$ for both parameters (i.e. obtaining an awkward banana-shaped posterior). Code to do this follows.

```
logposterior=function(p,sigma)
{
  a=1;b=1
  S=98;N=727
  dbinom(S,N,p*sigma,log=TRUE)+
```

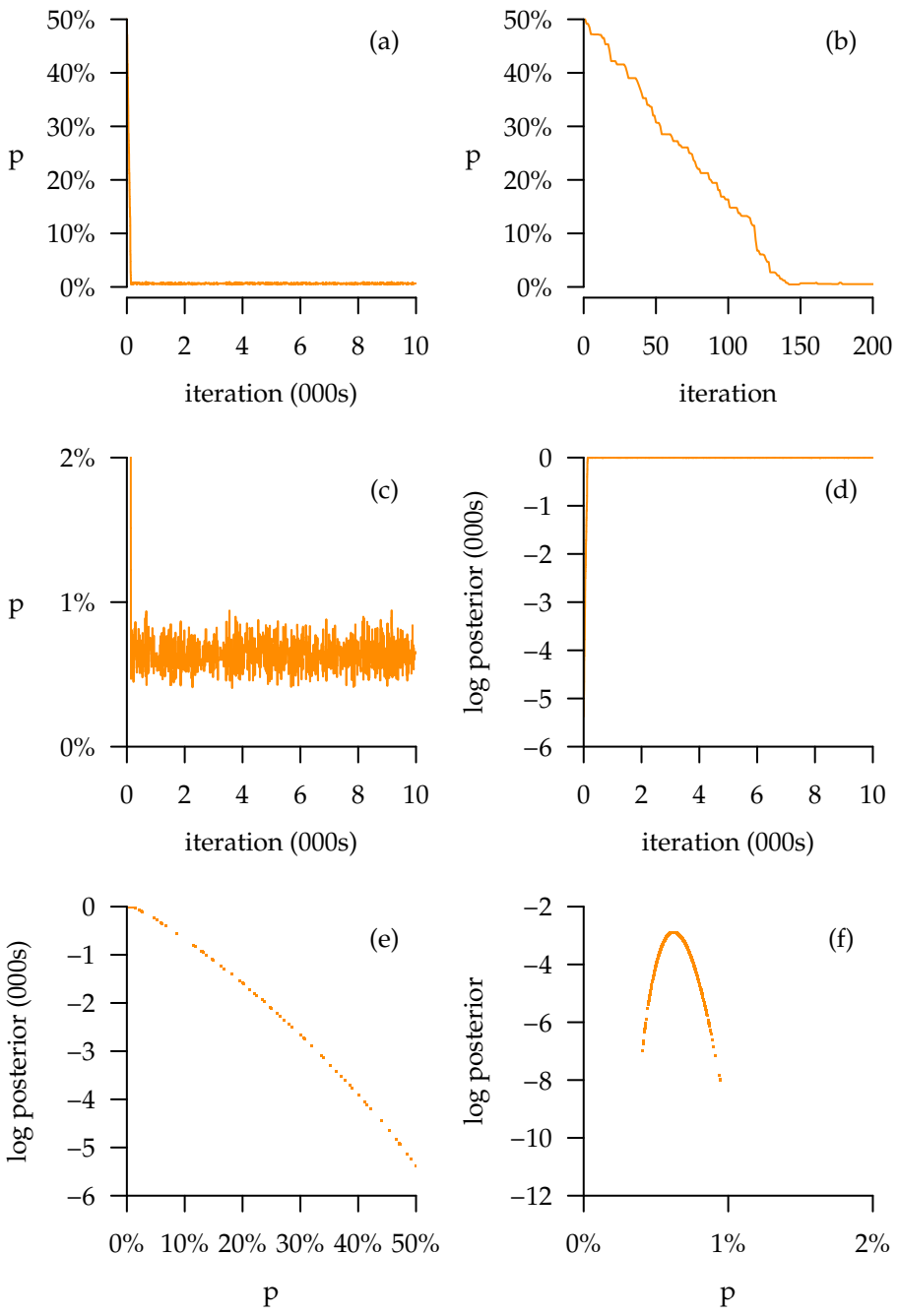

Figure 3.6: **MCMC output from Thai AIDS vaccine trial.** (a–c) show trace plot for p, the proportion vaccinated (zoomed in on earlier iterations for panel b, zoomed in on small values of p for panel c). (d) shows trace plot of log-posterior. (e–f) show log-posterior values for sampled values of p (zoomed in for panel f). Note that the sampler explores a very limited range of log-posterior values.

```
  dbeta(p,1,1,log=TRUE)+
  dbeta(sigma,a,b,log=TRUE)
}

set.seed(666)
p=0.5;sigma=0.5
logpost=logposterior(p,sigma)
MCMCiterations=10000
pstore=sigmastore=logpoststore=rep(0,MCMCiterations)
for(iteration in 1:MCMCiterations)
{
  if(iteration%%100==0)print(iteration)
  pold=p;sigmaold=sigma
  logpostold=logpost
  p=rnorm(1,p,0.01)
  sigma=rnorm(1,sigma,0.01)
  REJECT=FALSE
  if(p<0)REJECT=TRUE
  if(p>1)REJECT=TRUE
  if(sigma<0)REJECT=TRUE
  if(sigma>1)REJECT=TRUE
  if(!REJECT)
  {
    logpost=logposterior(p,sigma)
    logaccprob=logpost-logpostold
    lu=log(runif(1))
    if(lu>logaccprob)REJECT=TRUE
  }
  if(REJECT){p=pold;sigma=sigmaold;logpost=logpostold}
  pstore[iteration]=p
  sigmastore[iteration]=sigma
  logpoststore[iteration]=logpost
}
```

The resulting chain is plotted in figure 3.7. The mixing is poor, and Geweke's test indicates non-convergence (the Z scores are 3.1 and -3.9 for p and σ respectively, although the log-posterior's Z score is the very reasonable 1.5, indicating that it has converged and that the model is possibly over-parameterised). Repeating the analysis with a larger standard deviation (0.1 for each) improves matters (figure 3.8) but not perfectly. Running the chain for ten times as long and storing only every 10th value leads to satisfactory

mixing (figure 3.9) and the passing of Geweke's test ($Z = -0.00$ and -0.14).

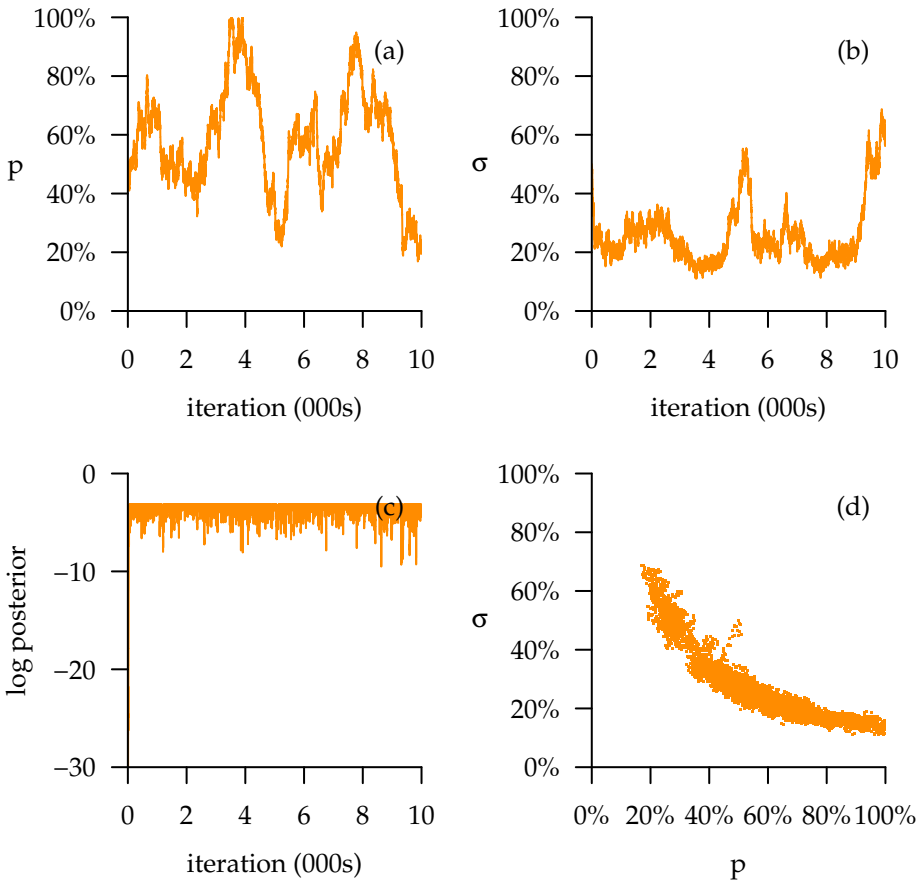

Figure 3.7: **MCMC output for Singapore influenza serology example.** (a–c) show trace plot for p, the proportion infected (a), σ, the sensitivity of the serologic test (b) and the log-posterior (c). (d) shows the joint distribution of the parameters. This is a poorly mixing chain.

3.5.11 Example: MRSA in a Singaporean hospital

The following example is based upon unpublished research with colleagues in National University Hospital, Singapore, on the impact of hospital-wide control measures on the number of cases of MRSA being treated per quarter. As the work is as-yet unpublished, the data presented here are synthetic, though modelled upon the actual data. The number of cases each quarter from the year 2000 to quarter 1 of 2012 was recorded, and the objective is to determine when the underlying trend changed and evaluate whether it could

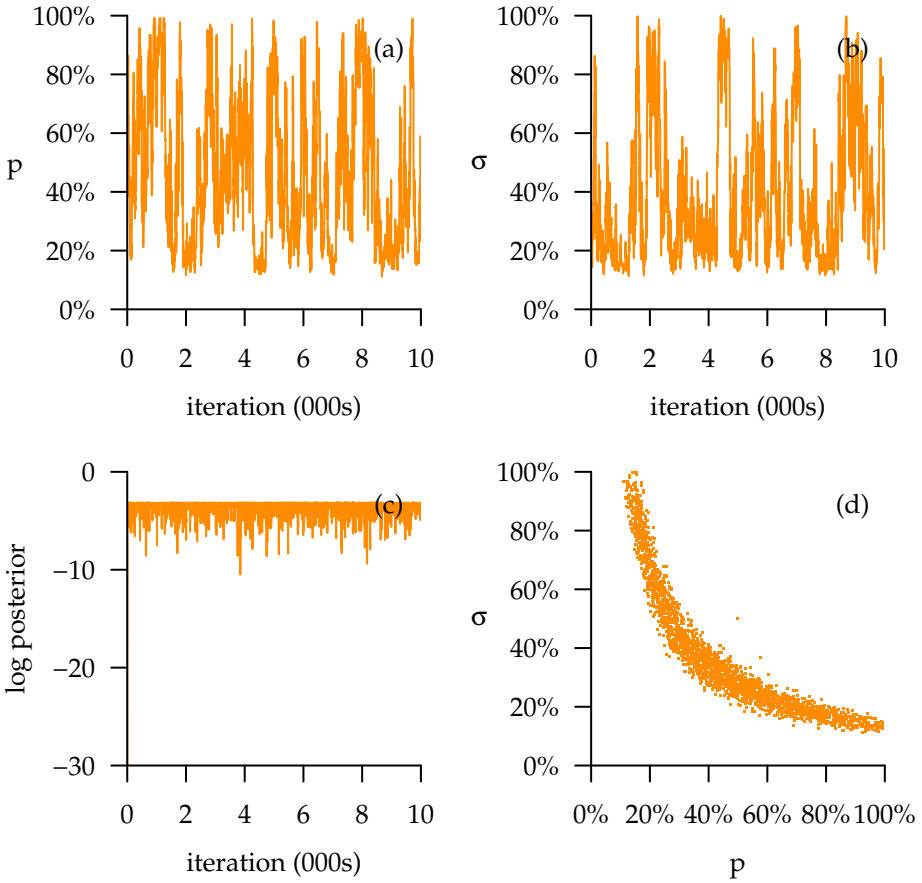

Figure 3.8: **MCMC output for Singapore influenza serology example.** As figure 3.7.

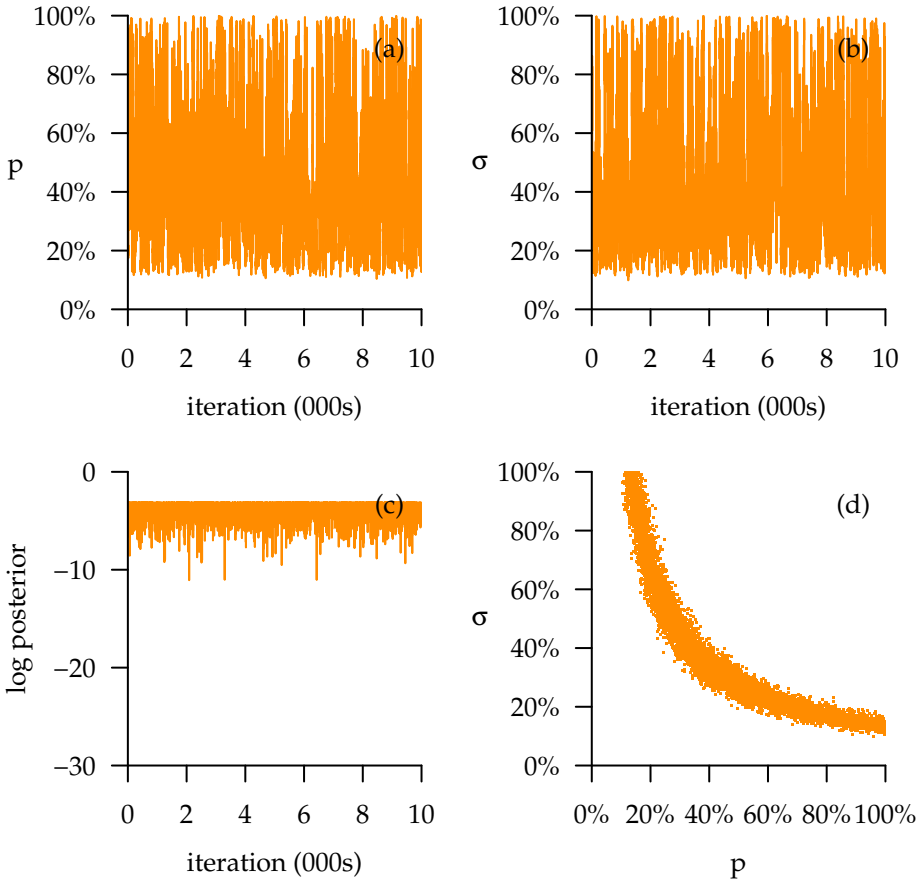

Figure 3.9: **MCMC output for Singapore influenza serology example.** As figure 3.7. Each 'iteration' is 10 proposals with only the 10th retained.

plausibly co-incide with the roll out of a bundle of control measures (which were extensive in scope and imaginative).

After some initial exploration, the model we decided to fit had the following form:

$$\log(m_q) = \alpha - \beta(\tau - q)\mathbf{1}\{q < \tau\} - \gamma(q - \tau)\mathbf{1}\{q > \tau\} + \epsilon_q \tag{3.13}$$

where m_q is the number of cases of MRSA in quarter q divided by the number of inpatient days across the whole hospital that quarter, τ is a change point, α the mean number of cases per inpatient day at the change point, β and γ control the slope before and after the change point, and ϵ_q is an error term, assumed iid normal with mean 0 and standard deviation σ (a plot of residuals suggests this is appropriate). The knowns are m_q and q, the unknowns $\alpha \in \mathcal{R}$, $\beta \in \mathcal{R}$, $\gamma \in \mathcal{R}$, $\tau \in [2000, 2012]$ and $\sigma \in \mathcal{R}^+$. We will try uniform priors on these ranges.

The code uses several functions, detailed and described below. The first, `meanify()`, takes two arguments—a list containing parameter values, and a vector of times (y CE)—and uses these to calculate the value of the mean (log) incidence at each time point, which it returns.

```
meanify=function(theta,q=seq(2000.125,2012.125,0.25))
{
  theta$alpha-
  theta$gamma*(q-theta$tau)*(q>=theta$tau)-
  theta$beta*(theta$tau-q)*(q<theta$tau)
}
```

The next function, `logposterior`, takes a list of parameter values as its argument, calculates the log-likelihood, sets the log-prior to 0 (a constant), and stores the log-posterior in the same list of parameter values, which it returns.

```
logposterior=function(theta)
{
  theta$logprior=0
  mq = c(
   0.00263, 0.00316, 0.00213, 0.00267, 0.00290, 0.00230, 0.00283,
   0.00281, 0.00293, 0.00272, 0.00283, 0.00298, 0.00307, 0.00290,
   0.00290, 0.00307, 0.00269, 0.00280, 0.00336, 0.00215, 0.00307,
   0.00329, 0.00357, 0.00427, 0.00368, 0.00321, 0.00388, 0.00343,
   0.00414, 0.00382, 0.00365, 0.00342, 0.00311, 0.00354, 0.00315,
   0.00396, 0.00357, 0.00342, 0.00334, 0.00322, 0.00290, 0.00289,
```

```
   0.00269, 0.00297, 0.00264, 0.00235, 0.00212, 0.00229, 0.00204
  )
  theta$loglikelihood=sum(dnorm(log(mq),meanify(theta),theta$sigma,log=TRUE))
  theta$logposterior=theta$logprior+theta$loglikelihood
  theta
}
```

The final supporting function, `mh()`, takes two lists of parameter values, and old and a current one, decides if any parameter values in the current configuration are illegal (if so, it returns the old configuration), if not, it then calculates the log-posterior at the current configuration, and performs a Metropolis–Hastings step, and then returns either the old or the current configuration.

```
mh=function(old,current)
{
  reject=FALSE
  if(current$tau<2000.125)reject=TRUE
  if(current$tau>2012.125)reject=TRUE
  if(current$sigma<0)reject=TRUE
  if(!reject)
  {
    current=logposterior(current)
    logacceptprob=current$logposterior-old$logposterior
    logu= -rexp(1)
    if(logu>logacceptprob)reject=TRUE
  }
  if(reject){current=old}
  current
}
```

The main code starts with a valid configuration (obtained by eye-balling), goes through 1 000 000 iterations, storing every 100th, proposing a change to each parameter independently at each iteration and accepting or rejecting changes to all simultaneously.

```
set.seed(666)
current=list(
        alpha = -5.5,
        beta  = 0.05,
        gamma = 0.056,
```

```
        tau   = 2009,
        sigma = 0.0001
       )
current=logposterior(current)
MCMCITS=10000
SUBITS=100
storage=list(
        alpha        = rep(0,MCMCITS),
        beta         = rep(0,MCMCITS),
        gamma        = rep(0,MCMCITS),
        tau          = rep(0,MCMCITS),
        sigma        = rep(0,MCMCITS),
        logposterior = rep(0,MCMCITS))
for(iteration in 1:MCMCITS)
{
  if(iteration%%100==0)print(iteration)
  for(subiterations in 1:SUBITS)
  {
    old=current
    e=rnorm(5,rep(0,5),c(0.01,0.01,0.03,0.5,0.0001))
    current$alpha=current$alpha+e[1]
    current$beta=current$beta+e[2]
    current$gamma=current$gamma+e[3]
    current$tau=current$tau+e[4]
    current$sigma=current$sigma+e[5]
    current=mh(old,current)
  }
  storage$alpha[iteration] = current$alpha
  storage$beta[iteration]  = current$beta
  storage$gamma[iteration] = current$gamma
  storage$tau[iteration]   = current$tau
  storage$sigma[iteration] = current$sigma
  storage$logposterior[iteration] = current$logposterior
}
```

Traceplots of the output and Geweke's diagnostic suggests convergence (figure 3.10). We can create traceplots of derived parameters, such as the mean (log) incidence in quarter 1 of 2010, to assess convergence too (panel g). Since it is space consuming to depict this for all time points, we might plot the mean curve for different iterations to get a rough feeling for how it changes for different parameter values (figure 3.11a). This can be extended

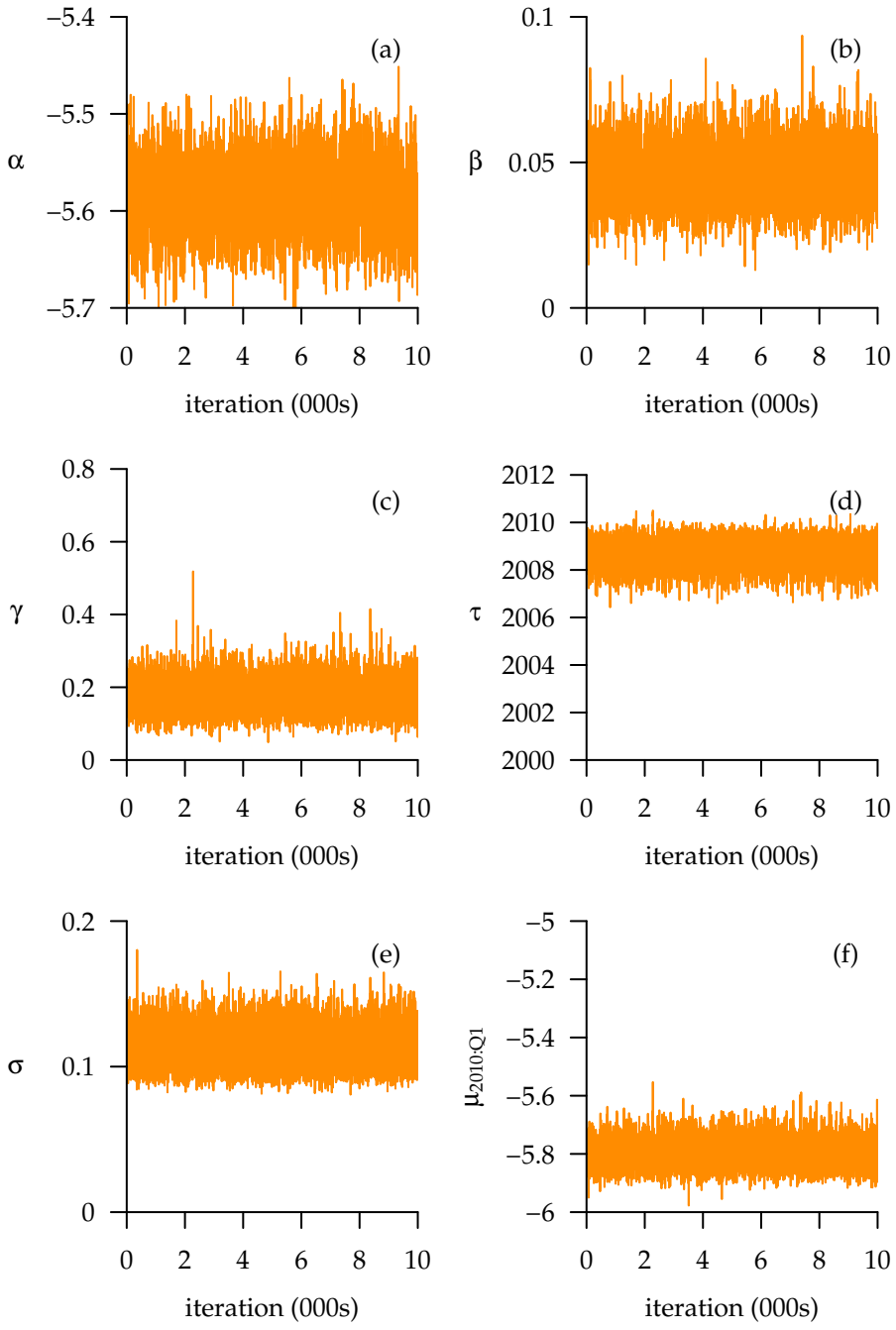

Figure 3.10: **MCMC output for MRSA in NUH example.** Traceplots are given for the five model parameters (a–e) and for the mean (log) incidence at one time point (f). The chain has been thinned by retaining every 100th draw of 1 000 000 prior to plotting.

to derive posterior mean and an uncertainty region for the mean curve over time (figure 3.11b) using the code below, which is an ideal way to summarise the results. The mean for τ, the change point, is Q4 2008 (95%I Q3 2007–Q3 2009) which is consistent with the roll out of interventions in the second half of 2008 (though the lack of a control arm or replication means we should not be overly confident in cause and effect).

```
CI=array(0,c(49,3))
for(j in 1:49)
{
  MU=meanify(storage,q[j])
  CI[j,]=quantile(MU,c(0.025,0.5,0.975))
}
```

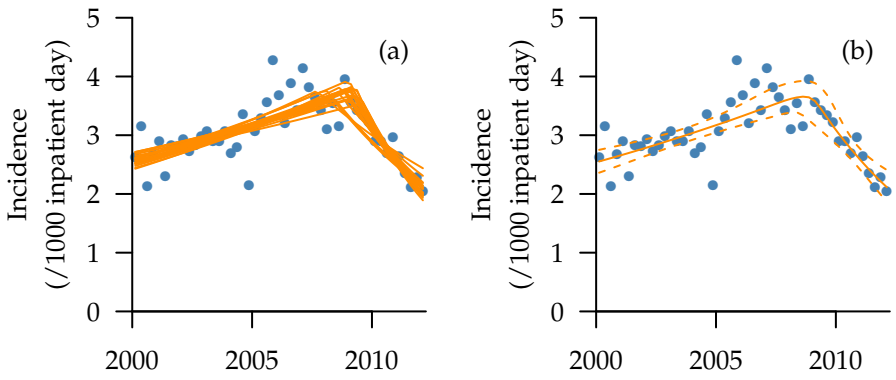

Figure 3.11: **MCMC output for MRSA in NUH example.** (a) shows 20 randomly selected draws from the posterior. (b) shows posterior median and 95% intervals.

3.6 BUGS and JAGS

It is important to be able to write an MCMC sampler from first principles as often you will face a problem that you cannot solve neatly, for example, for data that are not independent. However, writing code from first principles is not very efficient for most situations, such as when the data and parameters come from simple distributions.

It is based on the BUGS (Bayesian inference Using Gibbs Sampling) project started in 1989. It runs under Microsoft Windows, though it can also be run on Linux using Wine[1].

It was developed by the BUGS Project, a team of UK researchers at the MRC Biostatistics Unit, Cambridge, and Imperial College School of Medicine, London.

The last version of WinBUGS was version 1.4.3, released in August 2007. Development is now focused on OpenBUGS, an open source version of the package. WinBUGS 1.4.3 remains available as a stable version for routine use, but is no longer being developed.[2][3]

BUGS, short for Bayesian inference Using Gibbs Sampling, was developed in the late 1980s and early 1990s by a team of researchers from Cambridge and Imperial. It later became WinBUGS, though this has stopped being supported after attention switched to openBUGS in the mid noughties. Like its predecessors, openBUGS is free to download and use, and unlike its predecessors it can be used on other platforms than just Microsoft windows, though it is not as easy to use on other platforms and the documentation is poorer.

OpenBUGS can be run interactively, via a GUI (on windows), or via a command line interface.

JAGS is an alternative (that I prefer) "dialect" of BUGS. See later.

3.6.1 Why BUGS?

Imagine a spectrum of Bayesian software.

At one end is a routine built by someone else that is simple to use and only does one thing: Bayesian logistic regression, for example. The user specifies the data, perhaps a few other arguments, and the routine returns an answer. This is fantastic if you have a problem that needs exactly that, specific routine.

At the other end is a routine built by you that took a lot of effort to create and is for an obscure problem you might never encounter again. This is what you might need to be able to develop if you cannot find an existing routine that will solve your problem and don't wish to change the problem to make it more amenable to use in an existing routine (a standard consultant's gambit).

BUGS sits somewhere in the middle of this spectrum. It is not predesigned and limited to one particular problem, but it does not require completely developing a difficult routine from scratch. Instead, it uses (fairly) simple syntax to specify a model and data, and a simple interface to the MCMC configuration options, such as the number of iterations. The hard

work of constructing the sampler is then handled by the software. BUGS has the capacity to output to file for subsequent analysis (in R, say), and can be run from within R using an appropriate package.

3.6.2 Structure of BUGS/JAGS input

You typically create three files to use when running BUGS (or JAGS). The names are up to you, but it is worth choosing sensible ones like the following:

- `data.txt`: a file containing all the data used in the analysis. There is a specific format for this, described below. Note that BUGS and JAGS have a different format for input data.
- `model.txt`: a file containing a complete description of the model for data and parameters, again with a specific format. There are minor differences between the format used by BUGS and JAGS.
- `inits.txt`: a file containing initial values of the unknown parameters to be estimated. These are used to start off the MCMC sampler. If you use several chains, you would normally have different inits for each. The inits file is optional, and if you elect not to have one, the routine will (attempt to) start with "sensible" values.

With JAGS, and I believe openBUGS when run from the command line, you may also have a fourth file `script.txt` that tells the software what to do.

3.6.3 data.txt

The data should be in a list format, for example:

```
list(N=14,
     unit_type=c(1,1,1,1,1,2,2,2,2,2,3,3,3,3),
     infections=c(41,46,41,35,30,24,11,21,3,26,4,9,3,0),
     atrisk=c(92,92,87,95,71,102,90,88,29,161,27,42,31,8)
    )
```

3.6.4 model.txt

The format for a model description looks somewhat like an R script, but there are several differences. An `=` sign is not used to set one variable equal to another, only a left arrow `<-`. Tildes $\sim$ are used to denote stochastic relationships. Not all elementary functions you might expect from R are

supported. Distributions are denoted by 'dfoobar', where foobar is an abbreviated form of the distribution name. The normal distribution is specified by its mean and *precision*, τ, where $\tau = \sigma^{-2}$. I(believe i)mproper priors are not allowed, but can be approximated by using very large or very small numbers (e.g. `dunif(0,999999)` might approximate a $U(0, \infty)$ distribution).

An example model specification file might read as follows:

```
model{
  for(i in 1:N)
  {
    incidence[i]<-infections[i]/atrisk[i]
    incidence[i]~dnorm(mean[i],tau)
    mean[i] <- mu[unit_type[i]]
  }
  for(i in 1:3)
  {
    mu[i] ~ dunif(-1000,1000)
  }
  tau ~ dunif(0,10000)
}
```

Combined with the data file, this tells openBUGS that:

- infections and atrisk are vectors of length $N = 14$ and are known;
- their ratio is assumed to be normal with mean that depends on the unit type and common variance;
- there are three unit types and which data point corresponds to which type is known;
- the means are unknown and iid uniform;
- the precision is unknown and uniformly distributed *a priori*.

As a result, openBUGS will set up one or more samplers for the four unknown variables. Their initial values will need to be specified. This can be done by reading them from files.

3.6.5 inits.txt

This follows the same rules for specifying values as the data file does.

```
list(mu=c(0.3,0.2,0.1),tau=400)
```

If you are going to run multiple chains, you may wish to specify different initial conditions for them by having files named `inits1.txt`, `inits2.txt`, ..., and setting them to be dispersed relative to what you expect the posterior to resemble.

3.6.6 Running openBUGS in a GUI

This section is for windows users. For linux users I recommend using JAGS from the command line or R instead. First, open openBUGS in the usual windows way. Once it is open, there are a series of steps to go through to get openBUGS set up, tell it the model and data to use and give it instructions for running the MCMC routine.

You will probably find it useful to open a log to note any warnings that openBUGS tells you (unless your monitor is very narrow). After doing that, you should inform openBUGS that you wish to read information from files (the alternative is to select data or models with the cursor, which is okay for learning purposes, but dangerous for research (as you want to minimise the risk of mistakes creeping in and ensure reproducibility).

Following this, you are ready to input data and other information. This must be done in the following order: (1) model, (2) data, (3) specify the number of independent samplers, (4) either read initial conditions for each sampler from a file or let the computer generate them automatically. If there are errors, you may need to go through the entire process again. Note one annoyance I have encountered (every bloody time I use openBUGS): if you have the model or data files open in a text editor and are trying to read them in to openBUGS, you will not be able to save any edits you make to the file without changing the name or closing down the model specification box. (This may be a windows problem rather than an openBUGS one though.)

Once you have set up the samplers, you are ready to start them. However, you need to inform openBUGS of what variables it should track, as otherwise it will sample them all but not output any. This can be done before burn-in, in which case you can inspect the chains to determine appropriate burn-in length manually, or after a burn-in period. Usually I run the chains a while without tracking any variables and then switch on monitoring only after I guess the burn-in may have been sufficient. However, the screenshots below are for the first option. You must type in the names, exactly, of *all* variables you wish to track.

You may then get the sampler to start. Specify the number of iterations and amount of thinning desired. Progress updates will be posted to the same dialogue box. Once it has finished, you may select all variables you previously told openBUGS to monitor using a $\star$ symbol and create (ugly) trace plots

(please do not use these in any report) and output the samples via CODA.

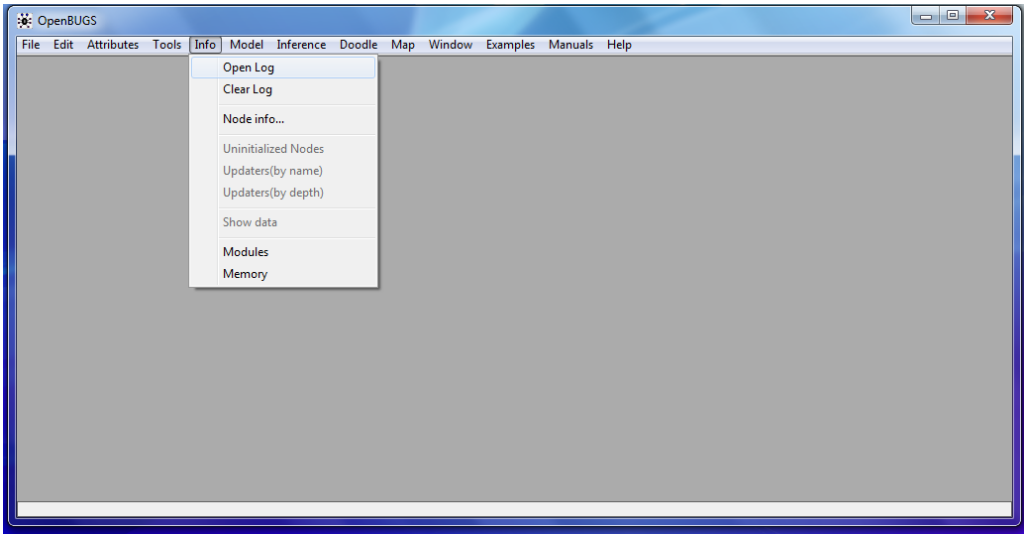

Figure 3.12: **Step 1:** open log to keep track of any warnings.

3.6.7 CODA

CODA is, confusingly, both a file format and an `R` package that can read files in CODA format and perform diagnostics on them. CODA output files contain one file per MCMC chain plus one additional file. The latter, the *index file*, lists the names of variables and two numbers i and j for each—i indicates the first iteration in each of the other files, and j the last, that correspond to that parameter.

As an `R` package, `CODA` allows you to input CODA files into `R` using the `read.coda()` function, run diagnostics (such as Geweke's diagnostic) and create trace plots, and export to other formats. There are four diagnostic routines are Geweke's, Gelman–Rubin–Brooks', Heidelberger–Welch's, and Raftery–Lewis'. The latter three do the following:

- Gelman–Rubin–Brooks. The function, `gelman.diag()`, takes multiple MCMC chains (assembled using `mcmc.list()`) and computes a "scale reduction factor" (SRF) that indicates how much the breadth of uncertainty intervals might shrink if the chains were run for longer. An SRF close to 1 is good.

- Heidelberger–Welch. The function, `heidel.diag()`, takes a single MCMC chain and tests the null hypothesis that the samples come from a stationary distribution, and if not, discards successive fractions of the

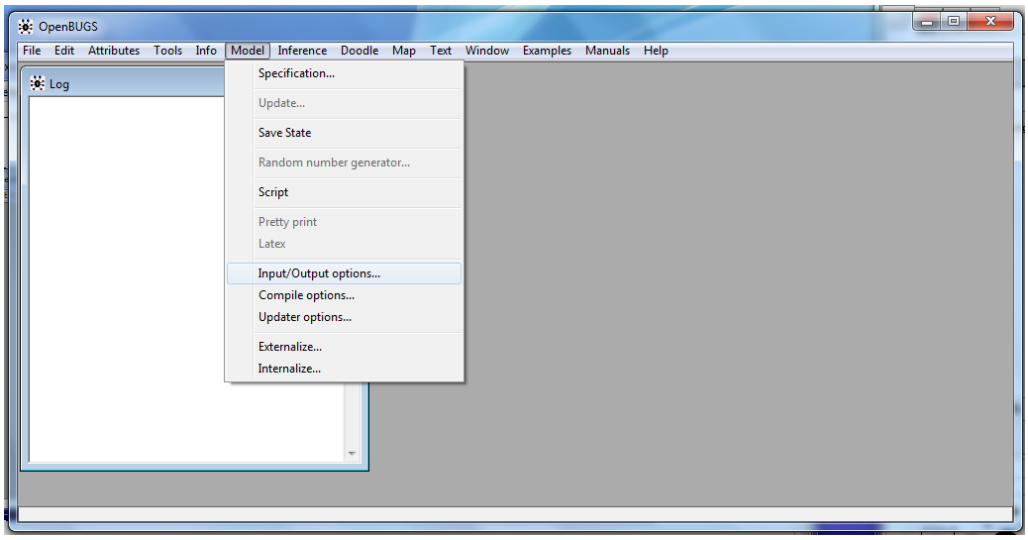

Figure 3.13: **Step 2a:** select input/output options.

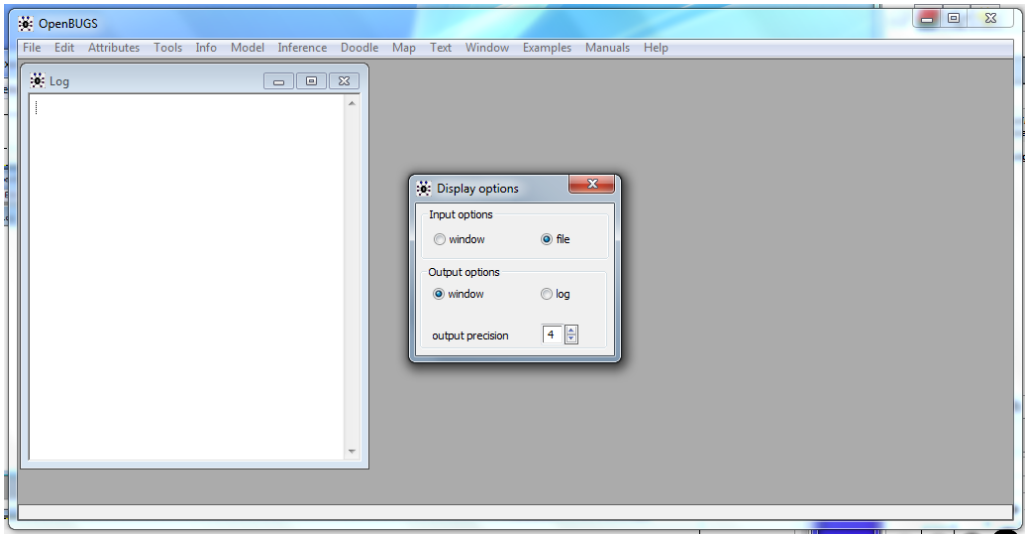

Figure 3.14: **Step 2b:** it is usually best to read data from a file, to ensure reproducibility, so select "file" as the input option. I find it fine to output to a "window" and save it via the menu. The default precision of 4 should be enough.

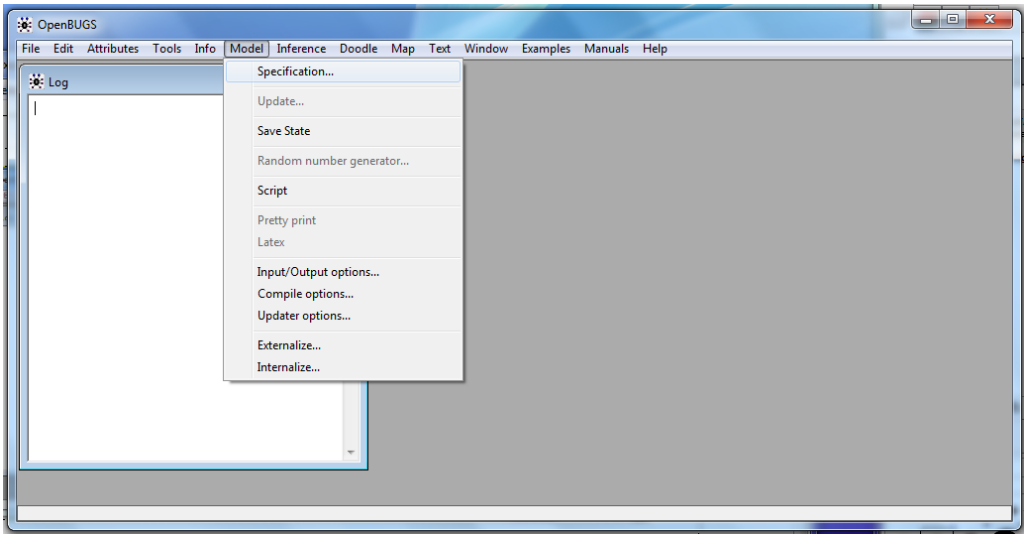

Figure 3.15: **Step 3a:** select the model specification option.

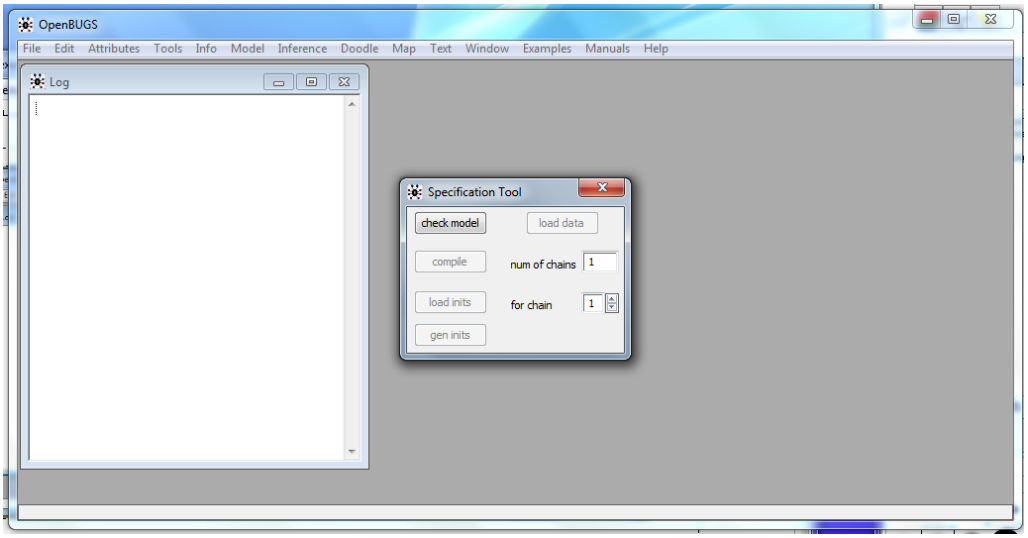

Figure 3.16: **Step 3b:** go through, in order, (i) checking the model, (ii) loading the data, (iii) compiling a number of chains, (iv) load initial conditions from a file and/or generate them automatically. These steps need to be carried out in order and have no errors.

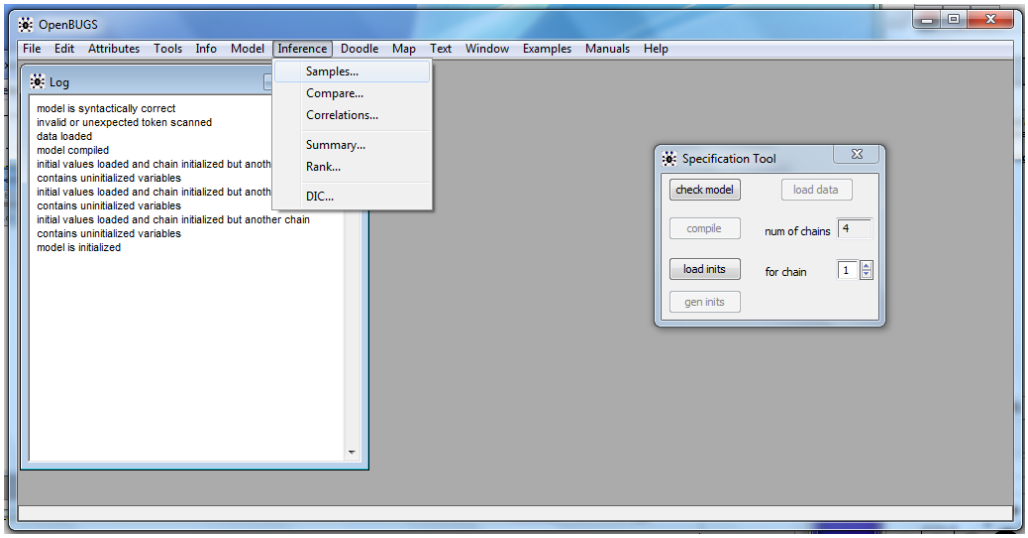

Figure 3.17: **Step 4a:** select samples from the inference menu.

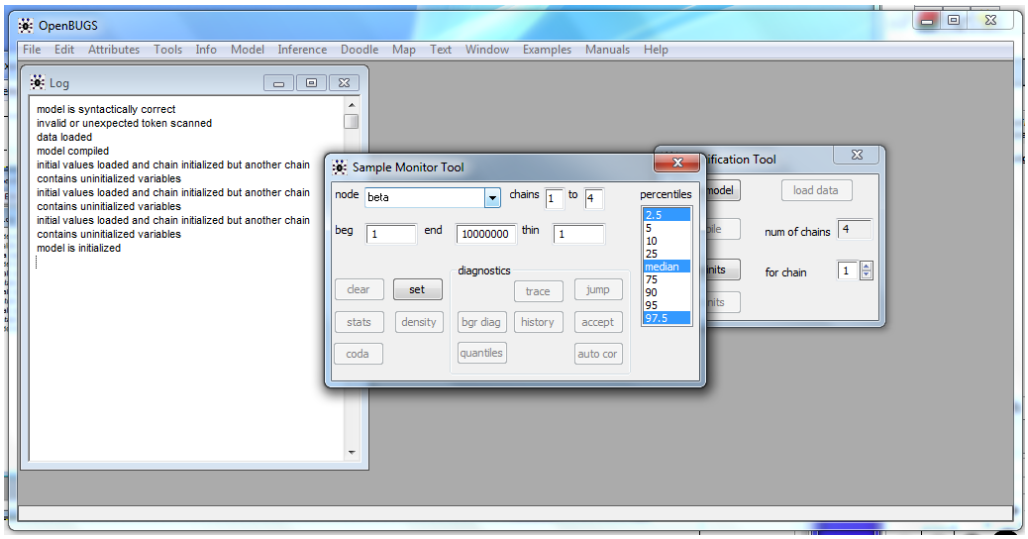

Figure 3.18: **Step 4b:** type each variable name (exactly) that you wish to track into the "node" field—this can be very tedius—and click the "set" option.

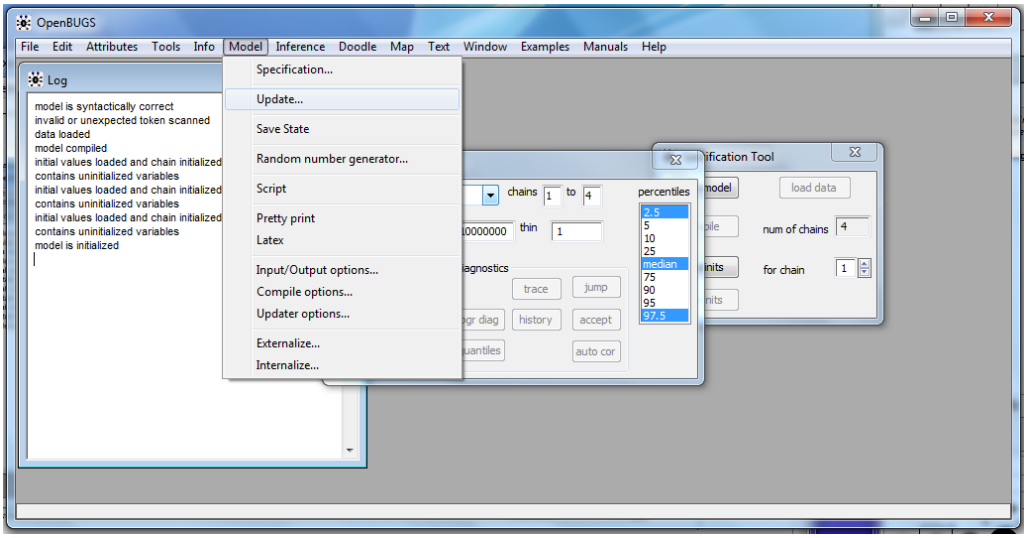

Figure 3.19: **Step 5a:** select the model update option.

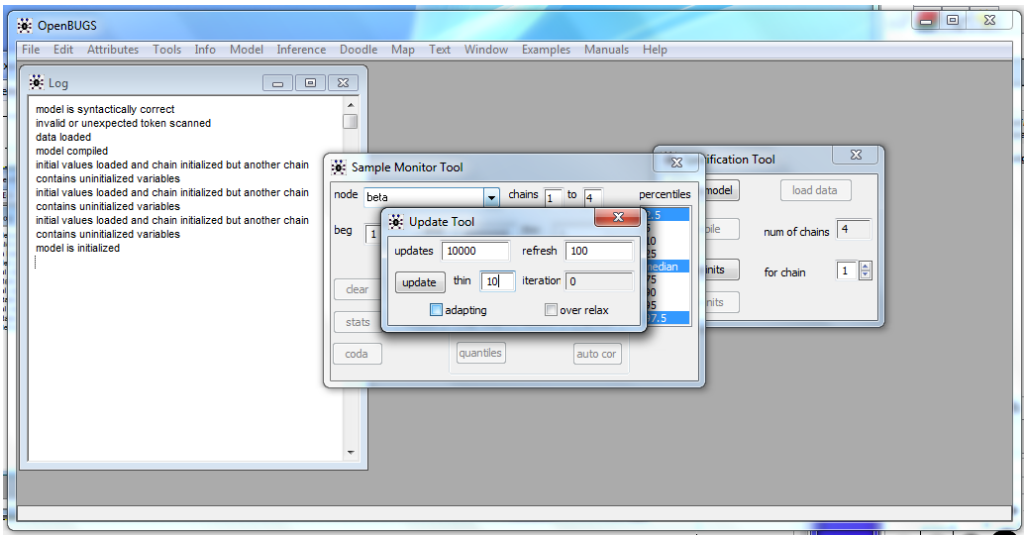

Figure 3.20: **Step 5b:** in the update tool, set the number of iterations desired, and any thinning required (e.g. the setting above would see 100 000 iterations in all with every 10th output.

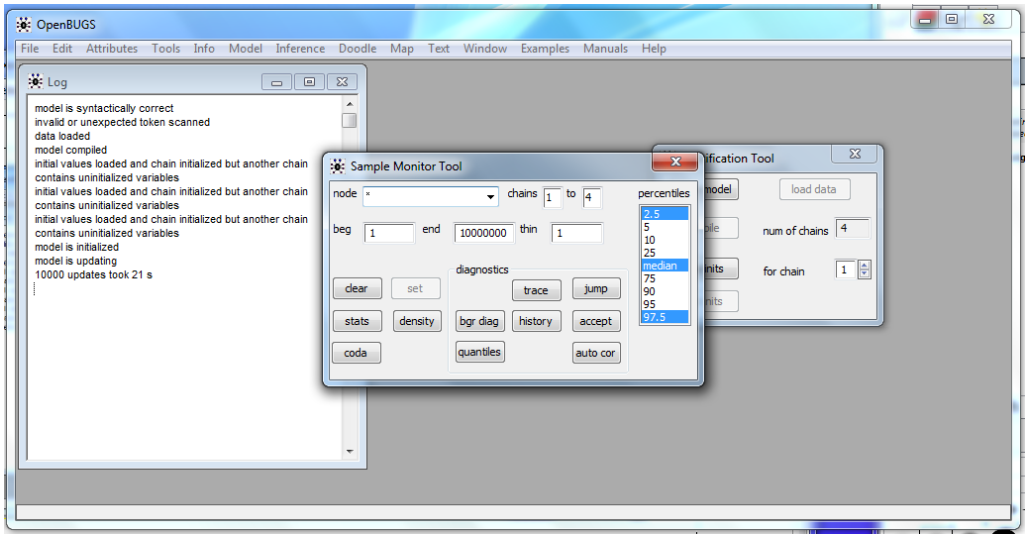

Figure 3.21: **Step 6:** select all parameters previously entered by entering an asterisk in the node field. You may then view a trace plot (via "history") and output the MCMC samples (via "coda") to files or windows (that can be saved from the file menu).

chain until it passes the test (and is thus a way to assess the length of burn in required).

- Raftery–Lewis. The function, `raftery.diag()`, takes a pilot run of an MCMC chain and calculates the number of iterations needed to estimate a quantile (such as the 97.5%ile) to a particular degree of accuracy.

3.6.8 Running JAGS

JAGS is Just Another Gibbs Sampler by Martyn Plummer. It is cross-platform, and hence is available on linux, mac, windows, and other major operating systems. It is also free. Unlike the windows version of openBUGS, JAGS runs from the command line (I find this far easier to do within a linux environment than on my wife's mac or the university's windows machines)—it can either take commands entered by the user, or a script. To run a script, type `jags foo.txt`, where `foo.txt` is the script name, in the desired directory. Of course, if the script is not in that directory you need to specify the directory in which JAGS can find it.

Here is an example of a JAGS script.

```
# Check model syntax
```

```
model in "model.txt"

# Load data
data in "data.txt"

# Compile with 2 chains
compile, nchains(2)

# Load inital values for first chain
parameters in "inits1.txt", chain(1)
parameters in "inits2.txt", chain(2)
initialize

# Start with 1000 update burn-in
update 1000

# Set nodes of interest
monitor a, thin(10)
monitor b, thin(10)

# Follow by a further 100,000 updates
update 100000

# Output
coda * , stem("CODA")

exit
```

This looks in the current directory for files containing the model description, the data, two sets of initial values, and then outputs to a series of files in the same directory with names starting with CODA. Typing the same commands interactively would yield the same results.

JAGS can also be run within R using the `rjags` package. Here is how you would do so for the Thai HIV trial. You will have a file containing the details of the model:

```
## model.txt ##
model {
  xv ~ dbin(pv,nv)
  pv ~ dbeta(1,1)
}
```

Then, within R, run the following:

```
library(rjags)
dataset=list(nv=8197,xv=52)
initialisation=list(pv=0.01)

jagmod=jags.model("model.txt",data=dataset,inits=initialisation,n.chains=1)
update(jagmod, n.iter=1000, progress.bar="text")
posterior = coda.samples(jagmod, c("pv"),n.iter=10000, progress.bar="text",thin=1)
```

This sets up the data and initialisations, usually read in separately from files in JAGS, then creates a JAGS model object within R by reading the model in, linking it to the data and initialisations, running one chain, updating the sampler by 1000 iterations (as burn in), then updating 10 000 iterations, storing `pv` in an object labelled `posterior`.

This is very useful as it allows you to exploit the built in simplicity of JAGS but fully within R, so the input can be created within R and the output manipulated within R.

3.6.9 BUGS example: influenza in SAF camps

Singapore's Armed Forces (SAF) stand always ready to protect the nation, and during the influenza pandemic of 2009 put various measures in place to ensure battle readiness of its troops. Fourteen distinct units, including 1015 men and women, took part in a serological study similar to the community study previously described (i.e. in which blood was taken at multiple time points, and antibody levels used to define a "seroconversion"). The primary difference is that the units received different levels of control measures, corresponding to their type, with regular units receiving a basic package of measures including social distancing, "essential" units (I think of these as being commandos, but was told they weren't) receiving additional measures, such as antiviral prophylaxis, and medical units receiving the same additional measures plus wearing personal protective equipment, such as face masks. The objective of the analysis is to relate unit types/interventions (the effect of which cannot be distinguished from inherent unit differences) to the outcome seroconversion (see reference [15] for full details of the study and analysis used).

The usual way one would tackle this problem is as a generalised linear model with binary outcomes for each soldier (seroconversion or not) and the unit type/intervention as a covariate. However, such a model is made inappropriate by the lack of independence between individuals serving in the same unit, who live together, work together, and probably party together

too. This means the variance in the estimators reported by a standard `glm` routine is smaller than it should be, and differences appear more "significant" than they should. However, the empirical proportion seroconverting should be approximately normal even without independence, and so an alternative approach is to treat the 14 empirical proportions seroconverting as the data and assume they come from normal distributions with different means for each unit type.

While this could easily be done in a frequentist framework, it turns out that a Bayesian analysis is more useful, for one objective of the study is to find the relative risk of seroconversion in different pairs of units. If the observed mean proportion seroconverting in the jth unit of type i is p_{ij}, and we assume $p_{ij} \sim N(\mu_i, \sigma^2)$ then the relative risk of seroconversion in unit type i, relative to k, is μ_i/μ_k. In a classical analysis, although a point estimate for this is trivially obtained, an uncertainty interval is difficult and requires approximations which are probably not justified as the sample size (now, 14) is small. In the Bayesian analysis below, both point and interval estimates are trivial.

The model for the data is described above, but we require also a model for the parameters. I would like these to be non-informative, and to do so have decided to set $\mu_i \sim U(-1000, 1000)$—though if the posteriors lay outwith the range $(0, 1)$ we should be concerned—and $\tau = \sigma^{-2} \sim U(0, 10000)$.

We will fit the model to these data using openBUGS. The model and data files are provided in sections 3.6.3 and 3.6.4. I will run four chains and use the following initialisation files:

- In `inits1.txt`: `list(mu=c(0.35,0.20,0.15),tau=300)`
- In `inits2.txt`: `list(mu=c(0.30,0.25,0.10),tau=900)`
- In `inits3.txt`: `list(mu=c(0.10,0.10,0.10),tau=100)`
- In `inits4.txt`: `list(mu=c(0.30,0.20,0.10),tau=400)`

I discarded the first 1000 iterations as burn-in and retained the next 10 000. I output the four chains to CODA, and then analysed them within R using the following commands:

```
library(coda)
c1=read.coda('codachain1.txt','codaindex.txt')
c2=read.coda('codachain2.txt','codaindex.txt')
c3=read.coda('codachain3.txt','codaindex.txt')
c4=read.coda('codachain4.txt','codaindex.txt')
cALL=mcmc.list(c1,c2,c3,c4)
plot(cALL)
```

which creates the (ugly) plot in figure 3.22.

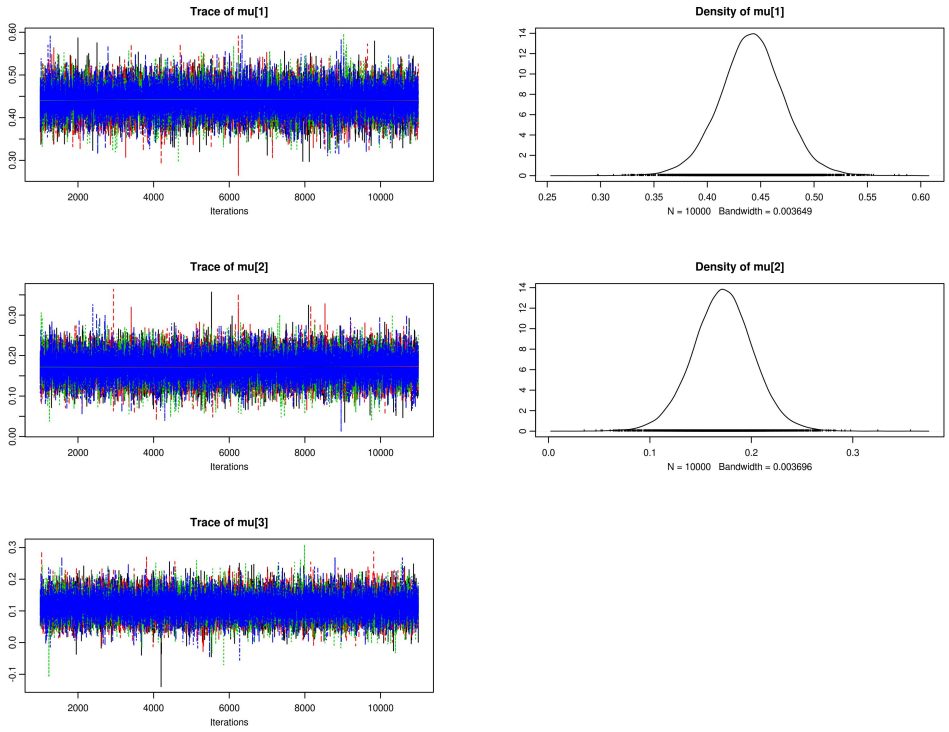

Figure 3.22: **Trace and density plots in CODA for SAF example.** Traceplots (left) show four independent chains (each with a particular colour). All look to have reached their stationary distribution. Density plots (right) aggregate the four chains.

We should run diagnostics on the MCMC output. The Gelman diagnostic is:

```
> gelman.diag(cALL)
Potential scale reduction factors:

     Point est. Upper C.I.
```

```
mu[1]          1          1
mu[2]          1          1
mu[3]          1          1
tau            1          1

Multivariate psrf

1
```

This indicates there is no benefit to running the chains longer. The Geweke diagnostic is:

```
> geweke.diag(cALL)
[[1]]

Fraction in 1st window = 0.1
Fraction in 2nd window = 0.5

  mu[1]   mu[2]   mu[3]     tau
-0.9790  1.5719  0.8428 -0.2161
```

(and similarly for the other three chains). All four Z test scores are reasonable. The Heidelberger–Welch diagnostic output is:

```
> heidel.diag(cALL)
[[1]]

      Stationarity start     p-value
      test         iteration
mu[1] passed       1         0.571
mu[2] passed       1         0.530
mu[3] passed       1         0.940
tau   passed       1         0.874

      Halfwidth Mean    Halfwidth
      test
mu[1] passed      0.441 0.000579
mu[2] passed      0.172 0.000514
mu[3] passed      0.115 0.000644
tau   passed    259.849 2.189520
```

This and the other three chains pass the stationarity test without additional burn in. They also pass the halfwidth test (which tests whether half the

width of the 95% for the mean of the posterior is less than 10% of the mean) though I don't find this so interesting.

The output can be summarised as follows:

```
mu1=c();for(k in 1:4){mu1=c(mu1,as.vector(cALL[[k]][,1]))}
mu2=c();for(k in 1:4){mu2=c(mu2,as.vector(cALL[[k]][,2]))}
mu3=c();for(k in 1:4){mu3=c(mu3,as.vector(cALL[[k]][,3]))}
tau=c();for(k in 1:4){tau=c(tau,as.vector(cALL[[k]][,4]))}
sigma=1/sqrt(tau)

summariser(mu1)
# etc
summariser(mu1/mu2)
# etc
```

The posterior mean proportion infected in the regular camps is 44% (95%I: 38–50%), in the essential units, 17% (95%I: 11–23%), and in the medical units, 12% (95%I: 5–18%). There relative risk for regular units is 2.7 times (1.8–4.0) that in essential units and 4.3 (2.4,9.2) times that in the medical units. There is no statistically discernible difference in infection rates between the medical and essential units ($\mathrm{p}(\mu_2 > \mu_3|\mathrm{data}) = 90\%$).

3.7 Other approaches

There are other techniques for obtaining samples from a posterior distribution, that are the subject of much current research. The Gelman lab has just launched a new package called STAN (which uses Hamiltonian Monte Carlo); other approaches are particle MCMC. But the techniques in this chapter will allow you to tackle all the problems you will face in the rest of the course, and probably most applied Bayesian data analyses.

The rest of the course will focus on using these techniques in genuine problems.